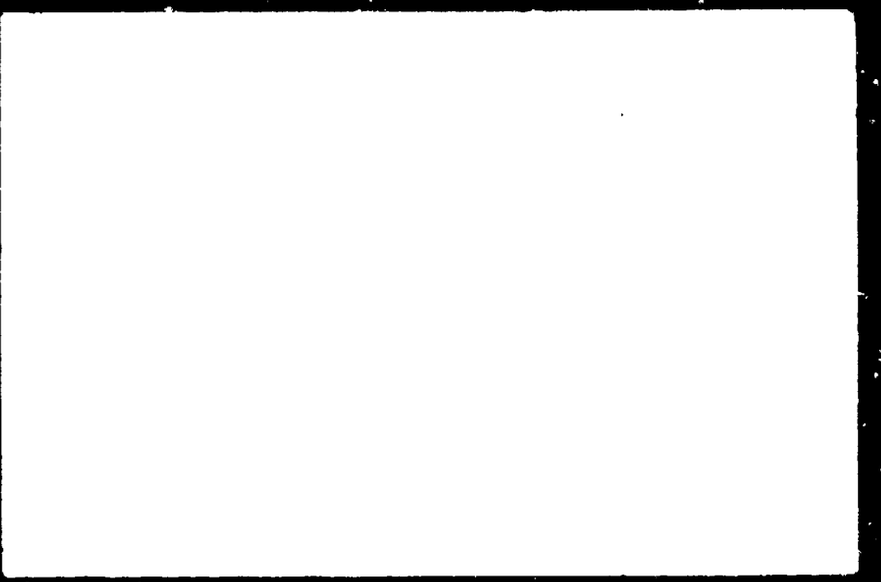

SERC

THE SOFTWARE ENGINEERING RESEARCH CENTER

GEORGIA INSTITUTE OF TECHNOLOGY
ATLANTA, GEORGIA 30332

A Unit of the University System of Georgia

USDR&E (T&E)

Software Test and Evaluation Project

GIT-SERC-87/03

Software Test and Evaluation Manual

Volume II

GUIDELINES FOR SOFTWARE TEST AND EVALUATION

in the

DEPARTMENT OF DEFENSE

25 February 1987

Prepared for

OUSDRE (T&E)
The Pentagon, Room 3E1060
Washington, D. C. 20301

Supported by

U. S. Army Missile Command
ATTN: AMSMI-PC-BFB
Redstone Arsenal, AL 35898-5280

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER GIT-SERC-87/03	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Software Test and Evaluation Manual, Volume II, Guidelines for Software Test and Evaluation in the Department of Defense		5. TYPE OF REPORT & PERIOD COVERED Final Technical Report 6. PERFORMING ORG. REPORT NUMBER GIT-SERC-87/03
7. AUTHOR(s) Software Test and Evaluation Project		8. CONTRACT OR GRANT NUMBER(s) BOA DAAH01-85-D-A005 DO 0008 and 0012
9. PERFORMING ORGANIZATION NAME AND ADDRESS Software Test and Evaluation Project Software Engineering Research Center Georgia Institute of Technology, Atl. GA 30332		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS U.S. Army Missile Command Redstone Arsenal, Alabama 35898-5280		12. REPORT DATE February 25, 1987 13. NUMBER OF PAGES 122 + viii
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Software Test and Evaluation Manual; Software Test and Evaluation Project(STEP); mission critical computer resources; software test and evaluation (T&E); risk assessment; operational and technical characteristics; software test tools and resources

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The Software Test and Evaluation Manual is a three volume reference set that provides checklists and guidance to Department of Defense components in the area of software test and evaluation for major Defense system acquisitions. These manuals are aimed at improving the test and evaluation of major systems through improved acquisition management and risk reduction procedures.

This manual addresses the structuring, planning, conduct, and evaluation of software tests throughout the acquisition process. Volume II is intended

(continued - over)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601 |

20.

for use by the Service Headquarters, Development Commands, Program Offices and Contractors, Development Test Agencies, and Operational Test Agencies.

Preface

The Software Test and Evaluation Manual is a three volume reference set that provides checklists and guidance to Department of Defense components in the area of software test and evaluation for major Defense system acquisitions. These manuals are aimed at improving the test and evaluation of major systems through improved acquisition management and risk reduction procedures.

Volume I, Guidelines for the Treatment of Software in Test and Evaluation Master Plans, is devoted to providing consistent guidelines for the preparation and evaluation of Test and Evaluation Master Plans (TEMPs) for major software intensive systems containing mission critical computer resources. It consists of a checklist i.e., a series of questions that are keyed to the major paragraphs of a TEMP, and an accompanying set of explanatory notes that provide brief commentaries on the questions and the significance of the possible responses to them.

Volume II, Guidelines for Software Test and Evaluation in the Department of Defense, addresses the structuring, planning, conduct, and evaluation of software tests throughout the acquisition process. Volume II is intended for use by the Service Headquarters, Development Commands, Program Offices and Contractors, Development Test Agencies, and Operational Test Agencies.

Volume III, Good Examples of Software Testing in the Department of Defense, is based on major programs which have benefited from many of the principles advocated by and detailed in Volumes I and II of this set. In particular, Volume III summarizes sample statements of work and contract text, program management hints, and other experience which has been derived from exemplary software testing efforts.

Table of Contents

Figures

Tables

CHAPTER 1

INTRODUCTION

The effectiveness of military missions depends on computer technology. As force multipliers, mechanisms for rapidly responding to changing threats, or tools for extending the data processing capabilities of individuals, computers -- and the software that controls them -- must function properly or else mission objectives are endangered. By virtually any measure of importance, software technology has become the critical risk factor in major Defense systems. The principal role of test and evaluation in the acquisition process is to reduce risk -- to evaluate the extent to which a given Defense system can be relied upon to fulfill its mission objectives in times of need. This manual explains in basic terms how to construct and carry out a software test and evaluation program.

Why a separate manual for software test and evaluation? In the first place, there are management and engineering imbalances between software and hardware that can only be rectified by specifying how effective test and evaluation of software is planned, carried out and evaluated. As long ago as 1974, a Defense Science Board Task Force studied these imbalances and concluded: "Whereas the hardware development was...monitored, tested and regularly evaluated, the software development was not." By 1982, the situation prompted the Secretary of Defense to direct the Military Services to "...give priority to the development of tools and techniques for testing of embedded computers and software." He further directed that "Testing of software should achieve a balanced risk with the hardware."

Another reason for writing such a manual is that it fills a gap between system level guidelines and technical textbook descriptions of software testing methodologies. Many acquisition managers are not software engineers. They have asked for an accurate but non-technical handbook that tells them where to start and how to tell whether or not they've left anything out. Many software engineers, on the other hand, have little experience with the structured Department of Defense acquisition process. They have asked for clear indications of how software technology fits into the overall system development.

Chapter 1: Introduction

The goal of these guidelines is to improve the test and evaluation of major systems through improved software acquisition management and risk reduction procedures. This manual is intended for use by all those in the acquisition community who are concerned with the risks of developing major systems that contain software. Thus, the primary audience consists of:

Service Headquarters,
Development Commands,
Program Offices and their supporting contractors,
Development Test Agencies, and
Operational Test Agencies.

It may also be of interest to Software Support Agencies and individuals within User Commands that become involved in requirements definition and evaluations prior to a new system's fielding.

The manual is structured to provide increasingly detailed discussions of software testing concerns that may arise at various decision-making levels in an acquisition. Basic definitions are reviewed and principles of test planning and evaluation are illustrated with a number of examples. These lead the reader to specific test methodologies and technologies as well as suggestions for where to find help and other resources. The reader should review the contents of this manual during the initial system planning stages for new acquisitions. This should help avoid some common pitfalls, such as structuring an acquisition that neglects software or delays decisions concerning software resources until time constraints make a progressive, systematic approach to software testing impossible to implement.

After an initial review, this document can serve as a reference manual answering questions concerning software test management, methodologies, and issues.

This manual and its companion volume "Guidelines for the Treatment of Software in Test and Evaluation Master Plans" have been designed to present a complete approach to software test and evaluation. These principles are consistent with existing Department of Defense policy and guidance as well as current Service regulations and standards. Furthermore, the testing process outlined in this three volume Software Test and Evaluation Manual has been routinely applied to major recent weapon systems acquisitions.

Chapter 1: Introduction

Despite its "How To..." appearance, this manual has some limitations. First, the manual does not adopt or suggest a cookbook approach to the software testing problem. Even when the reader makes a serious attempt to apply the principles outlined here, there is much left to be done. The bulk of the engineering analyses that are necessary for effective testing are highly specific to the system being developed, and no "generic" software methodologies can replace them. Second, rather than attempt a completely tutorial style or a completely technical presentation, this manual steers a middle course. Readers who have only system level software test and evaluation concerns as might be embodied in a Test and Evaluation Master Plan may rely on Volume I of this manual. Less experienced readers will find portions of this volume somewhat demanding. For those readers, a number of textbooks and other tutorial materials are available elsewhere. Finally, there are exceptions to the general guidelines presented here. The reader should not be frustrated if the approaches which are advocated in the remaining chapters do not apply intact to his or her specific problem. Specializing the generic approaches to the technologies or acquisition strategies requires engineering judgement and skill that a manual such as this can never replace. In such cases, the reader should be guided by the spirit and intent of the entire test and evaluation program.

The first step in applying sound software testing practices to system acquisitions is to determine the amount of software testing that is required -- that is, the extent to which software is a "risk driver" for the system as a whole. Chapter 2, Risk Assessment, outlines an approach for determining the extent to which a system is dependent upon software and the degree to which the system risk is inflated by software. It includes criteria for determining the extent to which a system contains mission critical computer resources or is software intensive. Risks resulting from operational requirements are analyzed in terms of both system level and software level requirements.

The next step in constructing an effective testing program is to define the specific goals or objectives of the test or test phase. Chapter 3, Operational and Technical Characteristics, provides a detailed treatment of the process of: (1) identifying thresholds for the software's required contribution to system characteristics, (2) identifying associated test and evaluation issues, and (3) measuring progress in the context of the system acquisition. Included are testable definitions for software quality factors, as well as typical critical test and evaluation issues that may be associated with software characteristics. Finally, the use of a Software Maturity Matrix to support "at-a-glance" information validity assessments during system/software reviews is discussed.

Chapter 4, Management and Schedules, includes the basic elements necessary for understanding and controlling the horizontal and vertical information flow associated with any major system acquisition. It provides an overview of the organizations normally involved in each acquisition and their roles as defined by appropriate policy documents. It also describes the variety of testing activities that contribute to a system's risk reduction and provides guidelines for their selection when investigating specific operational or technical characteristics. Finally, mechanisms for the management of information and scheduling of software-related test events are discussed.

The conduct of a software test is dependent on the test plan that guides the test. The utility of test results, on the other hand, depends on reporting them accurately and meaningfully. Chapter 5, Planning and Reporting, provides a roadmap for constructing effective software test plans at all levels, beginning with the Test and Evaluation Master Plan and continuing with the treatment of the DoD-STD-2167 test planning documents. The guiding software testing policy of Department of Defense Directive 5000.3 introduces the process of determining test objectives associated with critical test and evaluation issues, and then selecting appropriate test approaches for the test objectives. Reporting of test results and assessing the software contribution to system capabilities is also discussed.

Chapter 6, Software Test Tools and Resources, describes requirements setting procedures for software testing tools and methods for determining tool availability. Finally, options for building an organizational software test capability are presented along with an analysis of the associated risks.

This document is the second of a three volume set entitled the Software Test and Evaluation Manual. It has been prepared by the Software Test and Evaluation Project sponsored by the Deputy Undersecretary of Defense for Test and Evaluation (DUSD(T&E)). Volume I of the set, Guidelines for the Treatment of Software in Test and Evaluation Master Plans, provides consistent procedures and criteria for the preparation and evaluation of Test and Evaluation Master Plans. Volume III, Good Examples of Software Testing in the Department of Defense, cites useful practices drawn from major programs which have benefited from many of the principles advocated by the Software Test and Evaluation Project and detailed in Volumes I and II of this set.

CHAPTER 2

RISK ASSESSMENT

Risk is an element of uncertainty in the Department of Defense (DoD) acquisition process. Mathematically, risk is the probability or likelihood of failing to achieve a specific goal. In practice, the sources of risk that arise during system development are too complex to be treated with great mathematical precision. The goal of successful system acquisition is the effective management of risk. Test and evaluation (T&E) is a prime contributor to the process of assessing risks. T&E is an important and integral part of the overall job of risk management and the system acquisition process.

There are three principle kinds of risk that are usually of interest to decision makers and project management during a system acquisition:

* technical
* schedule
* budget.

Technical risk is mainly determined by uncertainties in the engineering process that may keep the system from meeting its technical specifications or may adversely affect overall system quality and performance. Schedule risk refers to all of the factors that may negatively impact the acquisition milestones. Budget risk refers to all of the factors that may cause unacceptable breaches of development cost allocations.

Sometimes, acquisition decision makers combine many kinds of risk into a single factor, called decision risk. Decision risk is the likelihood that an incorrect decision will adversely impact the attainment of system mission objectives. An example of decision risk is the likelihood that a decision to deploy a given weapon will result in the fielding of a system that is unsuitable for use by troops in combat. When accurate and complete T&E has been carried out and clearly reported, the decision maker has all of the information needed to reduce the decision risk. This is a critical step in risk management.

Chapter 2: Risk Assessment

Software often represents a significant source of risk in complex systems. Assessing software development risk through an effective software T&E program is often a key factor in determining and ultimately reducing overall system risks. Prior to assessing the software risk, however, it is necessary to identify the role that software will play in the system. For example, in some systems, software implements functions that are critical to meeting operational objectives. In other systems, software may represent a significant risk to the development effort (measured in dollars, for instance) even though it does not carry out any single function that directly affects system performance. This chapter outlines an approach for determining (1) the extent to which the system is dependent upon software and (2) the degree to which the system risk is inflated by software.

2.1 Identifying the Role of Software in the System

Two kinds of systems present software-related risks of major proportions:

* systems containing mission critical computer resources (MCCR)

* software intensive systems.

This method of classifying systems is not the only one, but it has been particularly useful in making early and accurate software risk assessments. As illustrated by Figure 2.1-1, systems for which software plays a key role can be either type described above, or both types. The finer points of such classifications are less important than the following: if a system meets any of the criteria described below for MCCR or software intensive systems, then it is very probable that software represents a main source of risk during system development.

Mission Critical Computer Resources

The term "mission critical" when applied to software is used in both formal and informal senses. In a formal sense, it means any software that falls under a legal definition of MCCR and therefore is subject to the highly managed acquisition process defined by DoD Directive (DoDD) 5000.1, Major System Acquisitions. When used informally, "mission critical" software simply means software that is essential to the successful performance of mission objectives, regardless of whether or not the system as a whole is classified as containing MCCR. If the system acquisition is managed under the auspices of DoDD 5000.1 or under Service specific policies and guidance, it is best to treat the software as a mission critical resource.

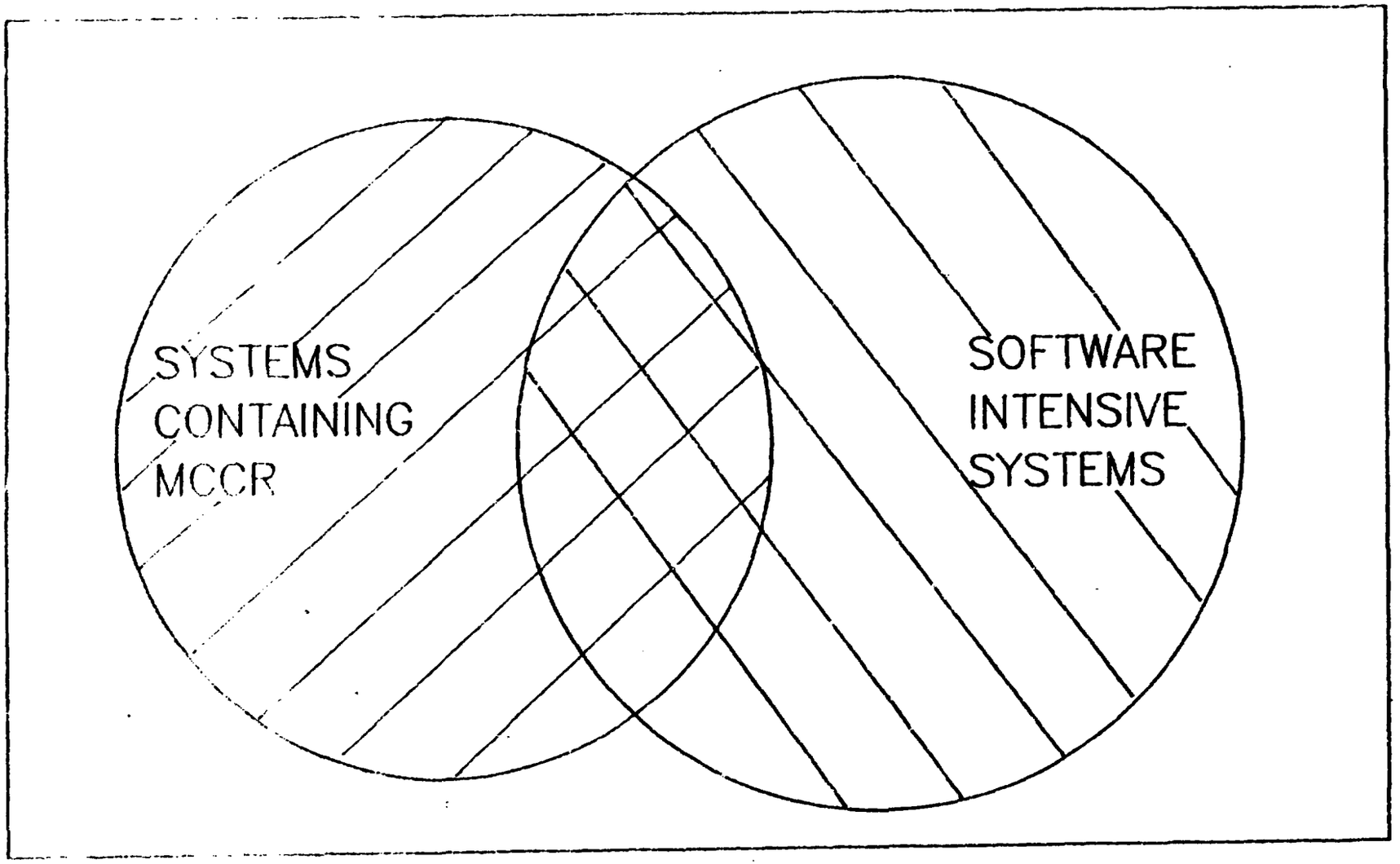

FIGURE 2.1-1: SYSTEM CLASSIFICATION IN TERMS OF SOFTWARE IMPORTANCE

Chapter 2: Risk Assessment

The Warner Amendment, 10 U.S.C. 2315, and Section 908 of the FY 1982 Defense Authorization Act define MCCR to include automatic data processing equipment or services whose functions are:

* Intelligence Systems
* Cryptologic Systems Related to National Security
* Command and Control of Military Forces
* An Integral Part of a Weapons System (i. e., physically a part of, dedicated to, or essential in real-time to performance of the mission of weapon systems; used for specialized training, diagnostic testing and maintenance, simulation, or calibration of weapon systems; or, used for research and development of weapon systems)
* Critical to the Direct Fulfillment of Military or Intelligence Missions including logistics systems which provide direct support to operating forces or provide direct support to maintenance of weapons systems (e.g. organic supply, software support facilities for weapon systems, etc.).

The original intent of these definitions was to exempt computers associated with weapon or intelligence systems from the General Services Administration procedures for acquiring data processing equipment. This was in recognition of the fact that computers intended for the applications noted above are frequently different in function, availability, and purpose from computers intended for business or research data processing.

Software in the systems listed above has potential importance to the satisfaction of mission essential operational requirements. In general, there are two types of mission critical software: application (or operational) software and support (or non-operational) software. Mission critical application software implements mission essential operational requirements. Mission critical support software implements software engineering functions and is used during the development and maintenance of the mission critical application software. When mission critical software exists in a system, the test program must be planned, conducted, and evaluated to ensure that system performance will not be impaired by improperly designed, implemented, or maintained software. Requirements for mission critical software can often be identified very early in the system acquisition process. If this determination is delayed too long, effective T&E will be impossible.

Chapter 2: Risk Assessment

Software Intensive Systems

The term "software intensive" is used to describe those systems in which software presents special sources of risk, regardless of whether or not the software implements mission critical functions. If a system does not contain MCCR software, it may still be software intensive and require a systematic and disciplined approach to testing.

Tests For Software Intensiveness

There are a number of tests that will determine whether a system is software intensive. The following examples will be discussed below:

* Do software costs dominate the total system development budget?
* Does the software contribute significantly to the operational and support costs of the system?
* Are large amounts of software required?
* Is software needed for successful system operation?
* Does the software integrate or interface a number of systems that must interoperate?

If the answer to any of these questions is "yes", the software intensive nature of the system gives rise to risks. In the first three cases, budget and schedule risks are implied. The latter two cases imply higher technical risks.

Development Cost Estimation

By most estimates, software costs (which are projected to reach $31.2 Billion for the DoD by 1990) constitute the major share of system development costs. An often-cited estimate of the Electronics Industries Association is that more than 80% of the cost of developing a typical weapon system is devoted to software development [EIA 84].

Nevertheless, determining the ratio of hardware development costs to the corresponding costs for software may be difficult to do in practice. One reason for this is that system acquisition costs are frequently quoted for the entire acquisition, not just the development phases of the acquisition. Even though software may represent 80% of the development budget, the average contribution of software to the total acquisition is less than 10% [Boe 81]!

This large difference is due mainly to the fact that total acquisition budgets include the costs of manufacturing and testing multiple platforms. Such costs are not generally applicable to software. When comparing costs, differences in the processes used to develop hardware and software must be considered. Development cost drivers for the hardware portion of a system are those associated with design, tooling, and production; where production costs are driven by the number of systems being produced and the cost of materials. Cost drivers associated with the development of the software portion of a system are also those of design, tooling, and production. In this case, however, production costs are negligible once the first software system is produced. Therefore, when comparing development costs, it is essential that software development costs be compared with the appropriate hardware development costs.

Another concern when estimating the cost of software development is ensuring that the total complement of system software is considered as opposed to the application or operational software alone. Non-operational software that is crucial if the system is to satisfy mission goals may include simulation, training, and diagnostic software, as well as support software. The costs associated with the development, testing, and maintenance of this software must be factored into the analyses whose purpose is to determine the impact of software on the system acquisition cost.

Life Cycle Cost Estimation

Sometimes, just estimating acquisition costs is not sufficient. It is common to move from one phase of a system acquisition to the next with immature software. Such software will be brought to a mature state during later, operational, phases of the system's life cycle. The cost of these activities (frequently classified as maintenance) does not get charged to the system's research and development budget. Therefore, estimating the so-called life cycle costs for the software may provide a better determination of how software intensive the system really is.

As with the development processes, the maintenance processes also differ for hardware and software. Thus, software maintenance costs should be derived separately from the corresponding hardware costs. Hardware maintenance cost drivers (e.g., availability of spares, logistics delays) are those associated with production. Production costs are driven by material costs and the number of systems being maintained or the number of parts being produced. On the other hand, since software maintenance accommodates changing requirements and the correction of errors by redesigning, recoding, and retesting software components and subsystems, software maintenance costs are heavily influenced by the costs of re-engineering.

Furthermore, the support software used during the development phase may not be available for maintenance activities. In this case, the life cycle cost estimates must incorporate the expense of acquiring support software specifically for use during the maintenance phase.

The Quantity of Software

The sheer amount of software that is required is often a significant source of risk. It is common to measure the quantity of software in terms of Source Lines of Code (SLOCs). Another common measure is the amount of computer memory in bytes or words taken up by the operational software. The latter measure, of course, excludes supporting software and may present a seriously misleading picture of the true magnitude of the software development effort.

Many studies indicate that software engineers produce SLOCs at an average rate of 400-800 per month (this average depends on numerous factors, including the nature of the application and the kind of programming language being used) [Boe 81]. Therefore, SLOCs is usually a reasonable estimator of schedule risk. In addition, most software development cost models also use the amount of software as the major cost driver. Less obvious risk factors that are often heavily influenced by the amount of software include those that reflect operational parameters. For example, industry wide data indicates a strong correlation between software size and the number of design flaws or "bugs" that are in the software. This, in turn, affects operational reliability.

In some acquisitions, software is acquired Commercially Off-the-Shelf (COTS). This is especially common in command and control applications, where many system functions can be implemented in generic decision support or communications software packages. The quantity of COTS software in the system may be a significant source of risk, even though many other risks are reduced by acquiring the software COTS. For example, the successful integration of diverse commercial software packages depends on uniform interfaces. If the software is acquired from different vendors, the interfaces may be highly incompatible. These concerns inflate the risk of utilizing COTS support software as well as COTS application software. Another example is the maintenance risks associated with a large number of commercial packages; this is especially relevant in a desk-top computer or workstation environment in which maintenance is normally supplied only through the vendor. Finally, the failure rate of a collection of independently developed packages tends to grow as a function of the number of such software packages in use.

Critical Software Components

A system is software intensive if its functions depend on software for its successful implementation. In this case, most of the technical risk associated with the system is concentrated on the software. Virtually all systems above the purely mechanical level that are currently in production, or are being planned, are software intensive in this sense. According to one recent study, 70% of the technologies, functions, systems, and actions identified in the Defense's long range plans require software [Red 84]. Many of these systems are not MCCR systems.

Table 2.1-1 contains estimates that can be used as a guide when assessing whether or not there is significant technical software risk associated with a new system development. These percentages represent the proportion of functions for each mission area listed that require software for successful implementation [Red 84]. For example, a communications function in a new system is virtually assured of being software intensive.

A system may, in addition, require software for successful operation even though no specific mission or functional area is addressed by the software. Examples include CAD/CAM and support software development, training and simulation, computer graphics and human interfaces, and decision support. A system having functional capabilities in any of these areas, or having significant interfaces with systems that provide these capabilities, is probably software intensive.

Command and Control	88%
Close Combat	78%
Fire Support	62%
Air Defense	89%
Intelligence and Electronic Warfare	83%
Communications	100%
Combat Support, Engineer, and Mine Warfare	48%
Combat Service Support	38%
Army Aviation	82%
Strategic Offense	100%
Strategic Defense	76%
Tactical Air Warfare	63%
Tactical Reconnaissance	88%
Electronic Combat	86%
Data Base Management	100%
Data Fusion	100%

Table 2.1-1: Percentage of Functions Supporting Mission Area that Require Software

Finally, it may be significant that the software requirements in a system represent "upgrades" to existing capabilities. The automation of previously manual functions, the introduction of new functions to previous software components, and the redeployment of existing software on new hardware all constitute significant sources of technical risk and indicate that the system is software intensive.

Software that Integrates Several Systems

Many systems depend crucially on the capability to communicate and interoperate with related systems. The area of communications interfaces demands special attention due to the fact that, with advances in technology, these capabilities are primarily embedded in software. Communications interfaces can be standard, Off-the-Shelf (OTS), or custom built. Experience with standard interfaces which have a specified predetermined protocol make their use low risk. At the other end of the spectrum are custom built interfaces and the high risks associated with attempts to define communications protocols that link multiple components or systems which are themselves in the process of being defined. In between these two extremes lie the OTS interfaces. System development risk is elevated when it is necessary to tailor OTS interfaces for the task at hand.

Identifying Critical Functions

A critical aspect of carrying out an assessment of technical risk is relating system functions and the software components which carry out those functions. The identification of these functions will form the foundation upon which the software test program will be built. The mission/function matrix, which lists the mission goals and objectives and relates them to system functions is a useful tool for identifying the critical functions implemented in software. Figure 2.1-2 represents a mission/function matrix for a battle management system containing computer hardware (H) and software (S) components. In this figure, an S signifies that the corresponding function will be implemented in software; an H indicates that the function will be implemented in hardware; and an S/H represents the intent to combine software and hardware to implement the function. Knowing which functions are composed of software allows the specification of software goals and thresholds, and the identification of associated software test issues. Plans for software testing, conduct, and evaluation revolve around these issues.

MISSION OBJECTIVES	FUNCTIONS				
	ACQUISITION	CLASSIFICATION	TRACKING	COMMUNICATIONS	NAVIGATION
LOCATE, IDENTIFY, & TRACK TARGETS IN THE PRESENCE OF ELECTRONIC COUNTERMEASURES	S(2)/H	S(1,2)/H	S(1,2)/H	H	S/H
DETERMINE GLOBAL POSITION & DIRECTION TO DESTINATION					S/H(3)
CARRY OUT ELECTRONIC COMMUNICATIONS				S(2,3)	
MANAGE BATTLE ASSETS & ASSIGN WEAPONS TO TARGETS UNDER HUMAN COMMAND CONTROL			S(4)		S

NOTES

1. MODIFY EXISTING SOFTWARE – NEW TARGET H/W FOR DEPLOYMENT
2. FUNCTION REQUIRES SOFTWARE
3. INTERFACE TO S/W INTENSIVE SUBSYSTEM
4. MODIFY EXISTING S/W TO INCLUDE NEW SEGMENTS

FIGURE 2.1–2: A MISSION/FUNCTION MATRIX

The success of a test program relies on the evaluator's ability to interpret test results at any point in the acquisition process in terms of system functions and mission objectives. If this is to be feasible, a complete tracing of mission objectives to system functions to hardware/software components to test cases must exist. The mission/function matrix is the first element of a chain of increasingly detailed specifications.

2.2 Assessing the Risks of Software Implemented Functions

Evaluations of major systems prior to deployment are concerned with determining the systems' operational effectiveness and suitability. Therefore, system acquisition risks are those of not satisfying effectiveness and suitability requirements. At the software level, this translates into the risks associated with failing to satisfy functional or quality requirements. This section will provide definitions of system operational effectiveness and suitability and will trace these definitions to system and software functional, performance, and quality requirements. The remainder of the section will discuss the risks associated with the achievement of operational effectiveness and suitability goals when the system functions are implemented in software.

DoD 5000.3-M-1, Test and Evaluation Master Plan Guidelines, defines operational effectiveness and suitability as follows:

> Operational Effectiveness. The overall degree of mission accomplishment of a system when used by representative personnel in the environment planned or expected for operational employment of the system considering organization, doctrine, tactics, survivability, vulnerability, and threat (including countermeasures and nuclear threats).
>
> Operational Suitability. The degree to which a system can be satisfactorily placed in field use, with consideration given to availability, compatibility, transportability, interoperability, reliability, wartime usage rates, maintainability, safety, human factors, manpower supportability, logistic supportability, documentation, and training requirements.

Relating Systems Engineering and Software Development

The achievement of operational effectiveness goals is dependent upon the specification and satisfaction of appropriate system and software, functional and performance requirements. The system engineering process is concerned with specifying system requirements, analyzing and refining those requirements, and finally, allocating them to subsystems and lower level components. These components are categorized in terms of their implementation medium: hardware or software. Once the requirements are assigned to components, the component development activities follow different tracks which are defined by military and DoD standards. For software, the principal standard that is applied is DoD-STD-2167, the Defense System Software Development Standard. This standard is required for use by all Services for systems which contain mission critical software. DoD-STD-2167 is based on the assumption that the system engineering process will produce a System/Segment Specification (SSS) and, as a minimum, a draft Software Requirements Specification (SRS) as a starting point for the software development process.

Tracing Operational System Requirements

The system functional and performance requirements specified in the SSS are designed to satisfy mission essential operational requirements and, therefore, trace directly to the DoD 5000.3-M-1 definition of operational effectiveness. Detailed software functional and performance requirements, specified in the SRS, support the system functions which have specified operational effectiveness requirements.

Quality Factor Requirements

Quality factors describe attributes of the system and the software that are required by the operational and technical objectives. Accompanying DoD-STD-2167 is a set of data item descriptions (DIDs) which define the contents and format of all deliverable documentation, including the SSS and the SRS. The SSS DID, DI-CMAN-80008, defines the contents of a paragraph for the specification of quality factor requirements. This DID provides for the inclusion of the following system level quality characteristics: availability, portability, reliability, maintainability, and flexibility and expansion. All of these, except flexibility and expansion which are actually aspects of maintainability, trace directly to the DoD 5000.3-M-1 definition of operational suitability.

The SRS DID, DI-MCCR-80025, also specifies required contents for a quality factors paragraph that is to include requirements for the following software quality characteristics: portability, reliability, maintainability, flexibility, usability, interoperability, correctness, efficiency, integrity, testability, and reusability.

In Table 2.2-1, the quality characteristics described above are listed. The X's indicate which of the documents (i.e., the DoD 5000.3-M-1 definition of operational suitability, the SSS quality factors paragraph, or the SRS quality factors paragraph) reference each quality characteristic. This table can be used to rapidly distinguish those characteristics that originate in the system level definition of operational suitability from those that are primarily software level concerns.

Risk Drivers for Effectiveness and Suitability

The primary risks associated with the achievement of the operational software's effectiveness goals are determined by the maturity of the functional area. These risks are naturally inherited by the software development. If the functions being implemented in software have been implemented before, then ignoring other factors, the current software development should be low risk. If the functions have not been previously implemented in either hardware or software, the expected risk of the software implementation is magnified since the definition of requirements and design cannot build on past experience. In this instance, the likelihood of false starts and dead ends increases significantly. These risks arise more frequently in new weapon systems due to the rapid growth in operational effectiveness requirements. These include requirements for increasing capacity as well as those derived from the necessity to counter more complex threats. For example, the number of simultaneous radar tracks required to be identified and stored has increased by an order of magnitude with each new generation of air defense systems. This has in turn become a significant risk driver for these systems.

The risks associated with the achievement of operational suitability goals are magnified by the failure to support the specification and measurement of progress with respect to software quality requirements. Quality requirements can be allocated to hardware and software according to a method similar to that used to assign functional requirements to hardware and software, but not without a consistent set of basic definitions. Many of the currently advocated definitions for the software quality factors are only remotely related to the system level definitions.

Quality Characteristics	Operational Suitability Definition	Quality Factors	
		SSS	SRS
availability	X	X	
compatibility	X		
correctness			X
documentation	X		
efficiency			X
human factors	X		
usability			X
integrity			X
interoperability	X		X
logistic supportability	X		
maintainability	X	X	X
flexibility		X	X
expandability		X	
manpower supportability	X		
reliability	X	X	X
reusability			X
safety	X		
testability			X
training requirements	X		
transportability	X		
portability		X	X
wartime usage rates	X		

Table 2.2-1: Relationship between Operational Suitability Definition and System/Software Quality Factors

Extreme requirements in such areas as reliability often necessitate the use of new algorithms and design techniques. Stringent operational suitability requirements for functions that are implemented in software are very high risk.

Finally, the software engineering technology being employed and its implementation in the selected support software can itself be a risk driver for the system software. The compiler is probably the most critical piece of support software used on a program. If the compilation process results in either incorrect or inefficient code, the operational software may not satisfy operational effectiveness and suitability requirements. Another kind of support software is automated software testing tools that provide assurance of correct operational software implementation. An error in such a tool or its underlying theory could mask an error in the operational software. Related risks may arise from seemingly unrelated factors. Suppose that an automated tool is used during the development process to perform requirements analysis including tracing the requirements to design and implementation components, and ensuring consistency between all interfaces. If the tool is proprietary to the developing organization and not available during the maintenance of the operational software, the risks associated with the support of that software as it evolves through its useful lifetime are increased.

The risks arising from software engineering methodologies or techniques, especially their impact on the operational software's effectiveness and suitability must be examined. When the technology is presented in the form of an automated tool, that tool should have been subjected to an evaluation process to ensure that its application will, in fact, provide a net benefit [E&V 84].

CHAPTER 3

OPERATIONAL AND TECHNICAL CHARACTERISTICS

Whether or not a system can satisfy a set of user needs and expectations is the major issue to be resolved by T&E. Required characteristics are the key indicators of a system's achievement of required capabilities. There are two types of required characteristics that are relevant: operational and technical. Accurate estimation of the operational characteristics allows the tester to predict the extent to which the final system will satisfy user needs and expectations. Determining the technical characteristics indicates the level of engineering quality in the system. In short, assessing technical characteristics reveals how well the system is being built, while the operational characteristics are used to determine whether the right system is being built.

A basic and fundamental activity of the tester is participating in the determination of the required characteristics of the system. System level requirements must be formulated to provide clear definitions of the required characteristics. This implies, for example, that requirements be testable; that is, there should be a set of characteristics associated with the requirements that are capable of being estimated by testing. Furthermore, threshold values must be established to determine minimum acceptable levels of achievement of required characteristics. Questioning whether or not the characteristics of the completed system will exceed threshold values gives rise to critical T&E issues to be resolved by testing. The effectiveness of the T&E program in establishing the overall worth of a system is often determined by how well this requirements setting process is carried out.

When software implements critical functions or otherwise provides a significant source of risk in the system, the contribution of the software to the required characteristics becomes a major factor in determining system worth. The previous chapter discussed in general terms the risks that are introduced by the presence of software in the system. This chapter will provide a more detailed treatment of the process of identifying thresholds for the software's required contribution to the system, identifying associated T&E issues, and measuring progress in the context of the system acquisition.

Chapter 3: Operational and Technical Characteristics

3.1 Determining Software Contribution to Required Characteristics

System requirements and the associated required operational characteristics are derived from the users' mission needs. The system requirements and engineering design parameters determine required technical characteristics. As a matter of engineering practice, the system-level requirements will be decomposed and allocated to subsystems, their functions, and ultimately to the system components. In order to be complete, this allocation should include both hardware and software components. At the system level, the test program is jeopardized by a lack of well-defined requirements. The same is true at the software level.

The allocation of system requirements begins at the most general levels in the decision documents and agreements that are used to manage the system acquisition. The principal document that drives the test program for a major system is the Test and Evaluation Master Plan (TEMP). Its use is directed by DoDD 5000.3, Test and Evaluation, and its format is specified in DoD 5000.3-M-1. It is within the TEMP that the system's operational and technical characteristics are delineated and related to specific test issues and objectives. When software is responsible for achieving the system's objectives, its contribution to the system operational and technical characteristics should also be evident in the TEMP.

DoD 5000.3-M-1 defines required operational and technical characteristics as follows:

> Required Operational Characteristics. Qualitative and quantitative system parameters approved by the user that are primary indicators of a system's capability to accomplish its mission (operational effectiveness) and to be supported (operational suitability).
>
> Required Technical Characteristics. Quantitative system parameters approved by the DoD Component that are selected as primary indicators of technical achievement of engineering thresholds. These might not be direct measures of, but should always relate to, a system's capability to perform its required mission function and to be supported.

Testability is a major concern when defining required software characteristics. Requirements that make perfect sense to software developers or even system end-users may not be susceptible to measurement or testing. For example, a characteristic such as "...the effort needed to perform..." is not testable in any obvious way. Such a definition leaves unresolved the questions of how effort is to be measured, under what conditions, and exactly whose effort is being measured. It is usually easy to rephrase such definitions as follows: "...the probability that a typical user will, under specified conditions, successfully perform..." In this definition the conditions of the test are more or less clearly defined (typical users and specified conditions) and the measurement criteria are explicit (a probability estimate derived from the statistical analysis of test data and events). In situations where more qualitative characteristics are used, the exact concept of probabilities can be replaced by less exact measurement criteria. For example, many measurements can be formulated to determine the extent to which a set of outcomes satisfies a previously defined list of criteria.

Thresholds define the minimum acceptable system performance required to successfully execute a mission. Despite their negative connotation (developers and users prefer to view the system from a more optimistic perspective, usually in terms of goals), threshold values are very important to the T&E process. In practice, goals can be sacrificed in the face of decreased budgets, shortened schedules, or unanticipated technical barriers. Thresholds, on the other hand cannot be negotiated away. Falling below an agreed upon threshold value indicates that one or more mission objectives will be impaired. Failure to meet or exceed a threshold for a required characteristic indicates a serious deficiency in the system.

DoD 5000.3-M-1 defines a threshold as follows:

> Threshold. A minimum level of performance required at a point in a system's life cycle such that the threshold at maturity equals the requirement. Achievement of the threshold should support a reasonable prediction that the system requirement will be met at maturity.

The identification of threshold values is an integral part of the process of setting overall system requirements. A by-product of this process is the establishment of thresholds for all hardware and software components that affect critical system capabilities. To the maximum extent possible, the requirements should "flow" from the system as a whole to the system components. A system acquisition is the end result of identifying a user need to counter a threat and translating that need statement into a set of mission objectives. However, as the system matures during engineering development and production, it may deviate in both capability and quality from what was originally planned. By defining the expected status of the system's maturity at key points in the development process, and comparing the observed maturity of the system with the predefined expectations, deviations can be identified and corrected early.

The requirements definition process results in thresholds for each of the system's required operational and technical characteristics. System requirements are implemented through some medium, usually either hardware or software. Thus, the thresholds set for each required characteristic must be appropriately translated into a meaningful requirement for the selected medium. If the capability of interest is implemented in hardware, the associated thresholds are translated into thresholds for required hardware characteristics. If the primary implementation medium is software, suitable thresholds for required software characteristics must be defined.

Suitable thresholds for required software characteristics must be the result of a comprehensive decomposition and allocation program which encompasses the operational effectiveness and suitability characteristics, as well as the technical characteristics. One such approach is outlined below.

Operational Effectiveness Characteristics

These characteristics are specifically related to mission objectives and are usually stated in terms of required system functions or capabilities. In many cases, individual system functions can be clearly associated with software components or subsystems. When this is feasible, the appropriate thresholds for required software operational effectiveness characteristics are inherited from the system level specification. In more complex cases, functional system capabilities define data processing "threads" that cannot be easily associated with a single software component. In these instances, a formal approach to allocating effectiveness characteristics and thresholds to specified software components should be used. Acceptable approaches involve defining data processing requirements using a model of the functional behavior, relating processing threads to overall software requirements, and carrying out the allocation steps in a systematic way within the model. Many of these software requirements definition methodologies are in common and widespread use [Alf 77, Ros 77, Tei 77].

Operational Suitability Characteristics

Table 2.2-1 provides a mapping of the DoD 5000.3-M-1 operational suitability factors to system and software level quality factors. This can be used as a starting point when determining required software operational suitability characteristics. Each operational suitability characteristic approved by the user is a potential required software characteristic. In cases where software terminology differs from DoD 5000.3-M-1 terminology, Table 2.2-1 and the definitions provided in Table 3.1-1 can be used to aid the threshold setting process. Definitions are also provided for operational suitability characteristics that are first introduced as software quality factors. Each of these should be considered for relevance on a system by system basis.

Once the appropriate software operational suitability characteristics or quality factors are identified, the threshold setting process is initiated. It may not be possible to specify or measure the achievement of quantitative thresholds for these software characteristics. In such cases, qualitative or comparative thresholds may suffice. In any event, just as in the setting of thresholds for effectiveness characteristics, software thresholds should be inherited from the system level thresholds. As described above, a number of widely used formal software requirements setting methodologies are available to aid in the allocation of threshold requirements.

Availability	Availability is the probability that the system will be in an operable and committable state at the start of a mission when the mission is called for at an unknown (random) time.
Integrity	Integrity is the probability that the system will perform without failure and will protect the system and data from unauthorized access.
Interoperability	Interoperability is the probability that two or more systems can exchange information under stated conditions and use the information that has been exchanged.
Maintainability	Maintainability is the probability that the system can be restored to a specified condition within a specified amount of time.
Reliability	Reliability is the probability that the system will perform as intended under stated conditions for a specified period of time.
Usability	Usability is the probability that users can operate the system under specified conditions without user error given they have received specified training.

Table 3.1-1: Quality Factors for Operational Characteristics

Technical Characteristics

Definitions for generic software technical characteristics that are also commonly referred to as software quality factors are provided in Table 3.1-2. As with the software operational suitability characteristics, these may or may not be direct descendants of system level characteristics. In any case, they should be considered as a starting point and examined for inclusion on a system by system basis. Once again the difficulties associated with specifying and measuring achievement of quantitative thresholds arise. Many times the simplest and most effective approach is to define the evaluation criteria for a given characteristic as a list of testable properties that, taken together, satisfy a more general software requirement. Quantitative thresholds can then be identified by referring to the extent to which the listed properties have been satisfied (e.g.,"... the system must satisfy 85% of the criteria..." or "...the current system must satisfy at least the criteria satisfied by the system it is replacing...").

As a general rule, common engineering practice should prevail in requirements allocation. However, software technology does not support quantitative specification and measurement to the same extent as hardware technology. The lack of a set of software "laws of physics" has resulted in the proposal of many controversial techniques for software quality prediction and measurement. When considering the use of software technology, it must always be remembered that there is no magic. Suggested techniques that are not accompanied by convincing evidence of effectiveness should be avoided.

3.2 Identifying Critical Software Test and Evaluation Issues

Once thresholds have been established for required software operational and technical characteristics, the resulting critical issues must be identified. These issues are simply the specific questions that must be answered by a test in order to assess the value of one or more characteristics. It is these critical issues that provide the basis for the selection of test objectives and appropriate testing methodologies. The identification of critical issues allows the concentration of test resources on areas where the most benefit can be realized. As stated above, not all required characteristics give rise to critical issues. If this were the case, the critical issues could not be used to prioritize test objectives and guide the development of an effective test program.

Correctness	Correctness is the extent to which the system conforms to its specifications and standards.
Efficiency	Efficiency is the ratio of actual utilization of the system resources to optimum utilization.
Expandability	Expandability is the extent to which the system capability or performance can be increased by enhancing current functions or adding new functions.
Flexibility	Flexibility is the extent to which system purpose, functions, or data can be changed to satisfy other specified requirements.
Portability	Portability is the extent to which system components can be transferred from one software system environment to another.
Reusability	Reusability is the extent to which system components can be used in other specified applications.
Testability	Testability is the extent to which the specified system operation and performance determine the conditions and criteria for tests.

Table 3.1-2: Quality Factors for Technical Characteristics

Chapter 3: Operational and Technical Characteristics

DoD 5000.3-M-1 defines critical issues as follows:

> Critical Issues. Those questions relating to a system's operational, technical, support or other capability, that must be answered before the system's overall worth can be estimated/evaluated and that are of primary importance to the decision authority in allowing the system to advance to the next acquisition phase.

Decisions made concerning the rigor and thoroughness of the test program as implemented for individual characteristics are based upon the inherent risks associated with achieving the specified thresholds. The risks of interest during the identification of critical issues are the same risks discussed in Chapter 2: technical, schedule, budget, and decision. In fact, the role of decision risk in the determination of critical issues is apparent in the DoD 5000.3-M-1 definition.

In practice, the critical T&E issues will be formulated by analyzing the required software characteristic that is being demonstrated, the demonstrated maturity of the system at that point in time, and the extent to which prior issues have been resolved. At times, the critical issues revolve around a clearly defined question such as whether or not a certain program can service all resource demands within a given interval of time. At other times, the critical issues involve complex interactions of components and can only be attacked indirectly.

The following paragraphs will outline considerations that should be taken into account when identifying critical software T&E issues that might be associated with operational suitability and technical characteristics. These questions tend to be more generic and susceptible to general discussion than the corresponding issues for operational effectiveness. Effectiveness issues tend to involve questioning specific aspects of system function. Nevertheless, the questions formulated below should be a model for all types of critical T&E issues.

The process of determining the specific questions of interest with respect to a given characteristic is one of stepwise refinement of the relevant requirements. This process will be illustrated using the operational characteristic of interoperability.

Example: Interoperability

Suppose that System X is being acquired and that one of its critical operational characteristics is that it be capable of interoperating with three existing systems: System A, System B, and System C. Further suppose that System X is required to interoperate with System A via a common data base coupled with compatible network protocols. Refinement of the requirement for a common data base may uncover a requirement that the data base be able to service concurrent requests correctly (e.g., without scrambling concurrent transactions, losing requests for service, or unfairly locking requests from service). Refinement of the requirement for compatible network protocols may surface a requirement for compliance with the International Standards Organization's Open Systems Interconnection (ISO/OSI) Model. Further refinement of the requirements for concurrency handling and support of the ISO/OSI Model could be expected to uncover additional, more detailed requirements. This stepwise refinement of the interoperability requirements is depicted in Figure 3.2-1. The criteria that determine interoperability between System X and System B, or System X and System C, could be entirely different than that used for System X when paired with System A.

The potential critical issues or questions derived from the interoperability requirements described above for System X and System A may include:

> Is the common data base employed by System X and System A capable of handling concurrent requests?
>
> Do the network protocols employed by System X and System A adhere to the ISO/OSI Model?
>
> What percentage of the time are both the common data base and the network available (i.e., what percentage of the time can System X and System A be expected to be capable of interoperating)?

Similar questions would arise concerning the capability of System X to interoperate with System B and System C, respectively. Any judgement of the worth of System X with respect to its operational characteristic of interoperability would require answers to these questions. Thus it can be seen that as system requirements are refined, each additional requirement may pose a new issue for investigation. These issues do not become critical issues, however, unless they lie on an important decision path.

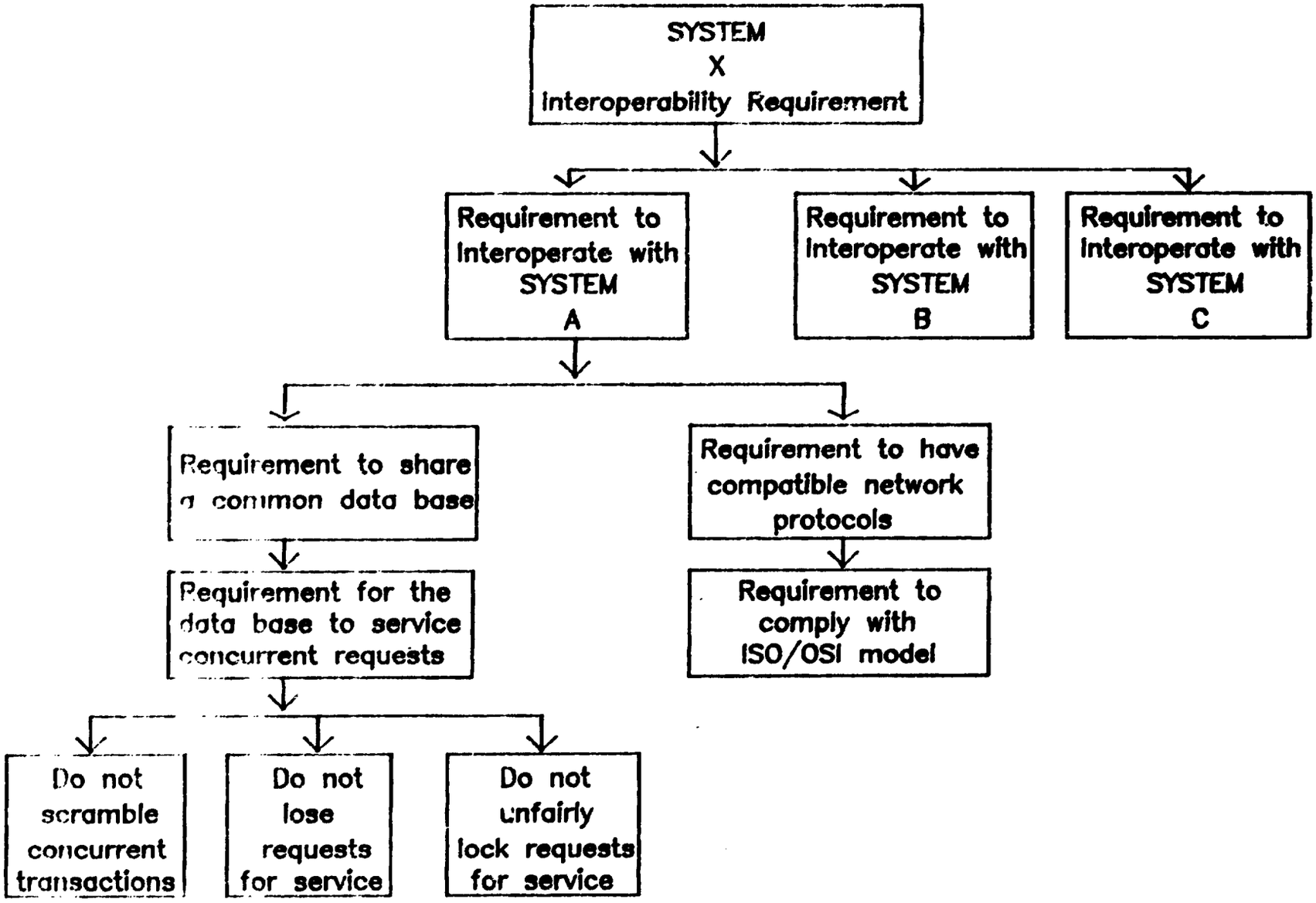

FIGURE 3.2–1: AN INTEROPERABILITY REQUIREMENT

Operational Suitability Issues

The remainder of this section will consist of typical issues or questions that may arise when the operational suitability characteristics of Section 3.1 are important to the system being acquired.

Availability

Software availability measures the likelihood that software implemented functions will be operational in times of need. There are two principal design mechanisms for ensuring availability: fault immunity and failure recovery. Fault immunity in a system guarantees that a function will be available by ruling out faults -- events that would cause the function to become inoperable. Failure recovery contributes to availability by ensuring that the software can transition from inoperable to operable states (after, for example, an operational failure has occurred) within specified time constraints. Among the most important availability detractors are the faults and failures that may occur in components outside the direct control of software designers. Contamination by hardware faults or failures in software systems with which the given software must interoperate is common and highly available software designs must provide adequate insulation from these sources. Another common detractor is the protocol that is required to bring the software to operational status (i.e., the procedures used to restart or re-initialize the system when failure recovery mechanisms are not sufficient). Typical software availability issues may include:

Do software functions remain available when presented with a closely spaced series of out-of-range input values?

If a software failure occurs, are its effects limited to the portion of the code containing the fault? Will the software recover to a failure-free state?

Do software functions remain available in the presence of specified hardware faults?

How are highly available software functions insulated from operating system failures?

Do "warm" or "cold" start protocols allow transition to full operational status within specified tolerances?

Integrity

Integrity measures the likelihood that the software and associated data are protected from unwanted access and manipulation. The widespread use of computer technology has provided new opportunities for unauthorized access of data and has thereby increased the risk of compromise of classified information. The principal design mechanisms for ensuring integrity include access control and data protection. Access control guarantees that system integrity is maintained by only allowing users to view authorized data and perform approved operations on that data. Data protection contributes to integrity by ensuring that the data is only received from expected sources, can only be interpreted by intended recipients and is, in fact, valid and consistent. Among the most important integrity detractors are trusted system components and multiplexed resources. System components that are trusted without investigation present, by definition, opportunities for compromise. Multiplexing -- or any sharing of resources -- allows multiple potential avenues for entry into the system. Another common detractor from system integrity is data corruption (i.e., the possibility that data may be unintentionally damaged or destroyed). Typical integrity issues may include:

Does the system password management protocol reduce the possibility of system access by unauthorized persons?

Is data "time-stamped" so that it can be validated prior to processing?

Does the software provide for controlled access to data and functions when requests can originate from multiple external sources?

Has evidence of trustworthiness been examined for those system components that have a trusted status?

Is data checked prior to storage or processing to make sure that it has not been altered by transmission over a noisy communications channel?

Maintainability

Software maintainability measures the likelihood that the software can be placed in an operational state when needed. This includes both the restoration of an inoperable function or capability (e.g., by pushing a "restart" button) and re-engineering of the software (e.g., to fix an error or to add a new capability), as appropriate. Software maintainability is greatly influenced by the software design and development process, and the capabilities existing in the maintenance depot. Maintainability is enhanced by software designs that build restore/restart capabilities into the system. The ease with which the software can be re-engineered is influenced by the original software development process. Software that is built using modern software engineering techniques lends itself to easier modification. Usually this kind of software maintenance is carried out at a Software Support Agency (SSA) facility. The SSA contributes to the software's maintainability to the extent that it supplies an adequate support environment and qualified personnel. Invariably, the maintainability of the software is limited by the operational environment's distance from the SSA and the logistic downtime that is required to recompile or restart the system. When on-site support is unavailable for the software, time delays associated with electronic communications or the physical transport of magnetic tapes place a lower bound on the minimum amount of time necessary to complete a maintenance action requiring software modification. Typical software maintainability issues may include:

Does the system include capabilities (e.g., cold and warm start functions) such that it can be restored to an operational condition from an inoperable state in the required amount of time?

Has the software been built using modular design techniques, accepted coding standards, and modern documentation practices?

Does the maintenance environment include the necessary compilation systems, testing tools, and documentation/management capabilities to allow software changes to be accomplished within the maintainability time constraints?

Do the software maintenance personnel have the proper training with respect to the application, software engineering, and the implementation language to allow the location and correction of an error within available time?

Can the modified software be delivered to the operational system to allow its installation in the required amount of time?

If necessary, can the system be restored to an operational state without taking it offline?

Chapter 3: Operational and Technical Characteristics

Reliability

A common error made when defining system reliability requirements is that of failing to consider the software's impact. If software is responsible for providing critical system functions, then the reliability of the software plays an important role in the overall system reliability. Software reliability measures the likelihood that software will perform as intended when called upon. The primary method of ensuring reliability is that of ensuring correctness of the software. Though the two characteristics are not equivalent, correctness is a great contributor to software reliability. Detractors from software reliability include the occurrence of any failures that affect software performance. This includes failures originating in the components of interest, other software components, and the hardware. As is the case with availability, contamination by hardware faults or failures in other portions of the software is common. If software is to satisfy extreme reliability requirements, its design must provide adequate insulation from these sources. Typical software reliability issues may include:

Has every software instruction been successfully exercised by some test case?

If a software failure occurs, are its effects limited to the portion of the code containing the fault?

Does the software perform as intended in the presence of operating system failures?

Does the software perform as intended in the presence of specified hardware faults?

Usability

Usability measures the likelihood that users can operate the system without error after specified training. Three principal determinants of system usability include the human factors considerations built into the software, the design of the hardware, and the qualities of the user documentation. Software human factors concentrate on the timely and consistent presentation of understandable information. Hardware designs must consider ergonomics and user skill levels. User documentation should maximize the ease of information location. The greatest detractor from system usability is the environment in which this characteristic is most critical -- the operational environment during extreme conditions. The stress introduced by situations requiring the non-drill use of military systems (e.g., engagement in battle) can reveal usability problems never suspected during experiments and evaluations. Typical software usability issues may include:

Do error messages guide the user through recovery procedures when needed?

Is there a roadmap to guide the use of the system documentation?

Does the system design minimize demands on the human user during stressful conditions?

Technical Issues

The process of refining requirements related to operational suitability characteristics to uncover software issues was illustrated above. If the refinement process is allowed to proceed to finer levels of detail, at some point the issues uncovered actually relate to technical software characteristics. An example of this already exists above: one of the reliability issues addressed the correctness of the software. Correctness is one of the technical characteristics presented below, along with its associated typical issues.

Correctness

Software correctness measures the extent to which the software conforms to its specifications. Thus, correctness is enhanced by the absence of errors, and detracted from by the presence of errors. Questions associated with the correctness of a piece of software, therefore, revolve around the process of determining correctness; in other words, testing. Typical software correctness issues may include:

Has the software been subjected to tests designed to reveal the presence of errors of specific types (e.g., computation errors, logic errors, data errors)?

Has the software been subjected to tests designed to demonstrate that errors of specific types are not present?

Efficiency

Efficiency measures the extent to which system resource utilization approaches optimum utilization. Real-time, embedded systems are sometimes accompanied by extreme efficiency requirements. For example, weight limitations may inhibit the ability to extend processing resources so the efficient use of available processors and memories is highly desirable. There are two principal contributors to the efficiency of a given implementation: the efficiency of the algorithms employed and the degree to which efficiency is supported by the development/maintenance environment. Theoretical analyses are available for determining the minimum number of processing steps required to implement solutions to basic mathematical or engineering problems. Taking advantage of the results of this field of study can contribute significantly to efficiency. Efficiency is further enhanced by utilizing support tools that have themselves been optimized for the efficiency of the resulting software. The primary detractor from application efficiency is the overhead resident in the use of underlying system functions. Every computer operation requires resources -- the careful design and selection of operators can reduce demands placed on scarce commodities. Typical software efficiency issues may include:

Does the software design employ basic utilities and data manipulation algorithms that minimize the combined utilization of scarce system resources (e.g., memory, storage, throughput, I/O channels)?

Does the software support environment include optimized code generators and timing/tuning tools?

Does the software implementation employ operating system functions in a manner that minimizes the combined utilization of scarce system resources?

Expandability and Flexibility

The extended useful lifetimes of major systems developed and fielded today necessitate the feasibility of modifying the system capabilities as the usage environment or threat changes. Expandability measures the extent to which these modifications can result in additional or enhanced system capabilities. Flexibility measures the extent to which the system can be modified to accommodate changing requirements. Similar to software maintainability, software expandability and flexibility are greatly influenced by the software design and development process, and the capabilities existing in the support environment. The use of modern software engineering techniques and the availability of an adequate support environment contribute to the expandability and flexibility of the software. These characteristics are further enhanced by designs and implementations that incorporate robust algorithms into the system. Limits associated with hardware capabilities and user workloads are the primary inhibitors of expandability and flexibility. Typical software expandability and flexibility issues may include:

Was the software designed and implemented in a modular fashion? Were programming standards that limit the impact of changes enforced during the software development?

Is necessary support software and documentation accessible?

Have basic system algorithms been designed to allow easy modification?

Can the hardware employed sustain the expected growth through the addition of boards, peripherals, etc.?

Are users capable of operating the system to take advantage of additional or changing functionality?

Portability

Portability measures the extent to which components can be transferred to other systems. This characteristic is usually of importance to software that is expected to satisfy the needs of a variety of users who employ a variety of hardware suites, or software that is expected to outlive its underlying hardware. In general, the only contributor to a software component's portability is its implementation. All other system interfaces or dependencies detract from or inhibit portability. These include required system libraries, the operating system, the implementation language, the hardware's instruction set architecture, and the machine level representations. Thus, typical software portability issues may include:

Are application interfaces consistent?

Do necessary math libraries provide adequate accuracy?

Have operating system dependencies been isolated and well documented?

Is the software written in a portable subset of the programming language?

Does the software use non-portable tests on collating sequences or exception flags?

Does a specified series of functions operate within the required time constraints?

Reusability

Reusability measures the extent to which components can be used in other applications. The amount of time, money, and human resources required to develop software has focused attention on the potential gains to be realized from employing, or reusing, individual software components in multiple systems. The reusability of a software component is enhanced by the use of modern software engineering techniques and robust algorithms, as was the case for expandability and flexibility. Of special and unique importance to the reusability of software is the availability of evidence of the component's behavior under a variety of conditions and scenarios. As expected, detractors include all hardware or operating system dependencies. Typical software reusability issues may include:

Has the component been built to allow its extraction from the remainder of the system? Is the software modular? Are interfaces parameterized? Is separate compilation supported?

Are test history information and results available to provide confidence that the software will perform as intended in the new environment?

Have basic system algorithms been designed to allow easy modification?

Have hardware and operating system dependencies been isolated and well documented?

Testability

Testability measures the extent to which proper system behavior can be determined. The primary factors that influence the ability to test software are the specification of software requirements, the organization of software components, and the availability of appropriate data extraction and reduction tools. Without traceable, consistent, and adequate software requirements to determine if software behavior is acceptable, the testing process is doomed. When the software components are hierarchically organized and traceable to requirements, software testing can follow a progressive process where each test builds on the results of tests that have already been conducted and analyzed. Building software to include instrumentation or hooks further supports testability. Testability is inhibited by the impact that the testing activity has on system behavior and the increase in system size and complexity when test capabilities are added. Typical software testability issues may include:

Is the expected behavior of the software (e.g., functional operation and performance) described in an unambiguous manner?

Can software components or subsystems be examined independently for testing purposes?

Do capabilities exist to control software execution and gather all data needed to determine software behavior?

What impact does the execution of test data extraction routines have on the true software timing profile?

How have increases in the complexity of the software due to the inclusion of instrumentation been accommodated by the testing process?

The typical issues described above are intended to suggest potential critical issues. However, any decision concerning criticality must be made on a system by system basis. Once critical issues are identified, related test objectives are defined and suitable test methodologies are selected as described in Chapter 5. It should also be noted that critical issues do not necessarily remain constant throughout the life of a system acquisition. Thresholds set for operational and technical characteristics, as well as their associated risks, may change. This topic will be addressed in the following section.

3.3 Determining Software Specification and Demonstration Milestones

The previous sections have explained the importance of setting thresholds for the software operational and technical characteristics and have provided typical issues that may result from those characteristics. This section will discuss the acquisition cycle and the information available to support the specification and evaluation of thresholds during selected phases of that cycle. These concepts provide a context that can be used to judge information received during an acquisition and to ensure that expectations of progress are realistic.

Figure 3.3-1 depicts the generation of a system as a cyclic process which begins with the identification of an operational threat from which mission essential operational requirements are derived. A variety of concepts may be envisioned for the accomplishment of the mission objectives, but eventually, a single conceptual system is selected and described by a set of system requirements. These requirements are reflected in a system specification and then partitioned into functional capabilities for implementation in software and hardware. As detailed requirements are further refined and amplified, designs for individual software and hardware components emerge. As the components are built, tested and integrated, a preproduction system evolves and system development testing (DT) is performed. Following successful completion of DT, the preproduction system is subjected to operational testing (OT). Finally, production quality systems are built and deployed. The cycle begins again as knowledge, gained from operational experience with the system, is factored into plans for new systems that may be developed to counter a future threat.

Another view of this process is presented in the Software Maturity Matrix (See Figure 3.3-2). This matrix is a tool that can be utilized by decision makers when assessing software-related information during the different phases of the acquisition cycle. DoD 5000.3-M-1 defines a mature system as follows:

> Mature System. A system meeting the minimum essential DoD Component-approved operational, technical, and quantity requirements baseline for full and complete fielding of a weapons system. To be mature, the system must have achieved its reliability thresholds and be fully maintained and supported in accordance with the DoD Component's maintenance concept.

Principal drivers that influence software maturity during a system acquisition include the extent to which requirements are known, experience with applying automation to similar applications, completeness of the software development effort, and the degree to which the software has been exercised, tested, and perfected.

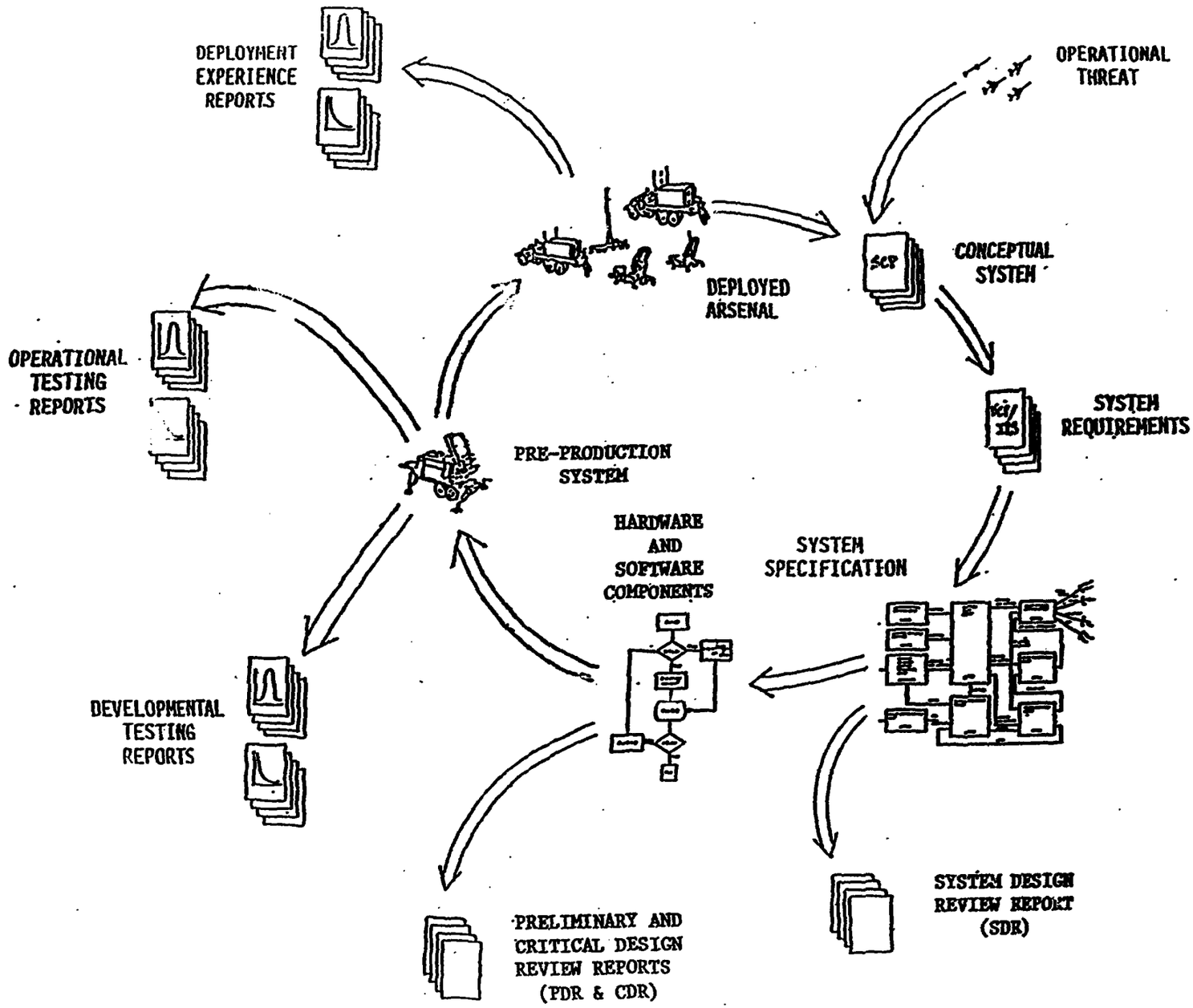

FIGURE 3.3–1: ACQUISITION AS A CYCLIC PROCESS

Acquisition Milestones	Program Initiation 0	Alternative Selection I	Intend to Deploy II	Production Decision III
Acquisition Phase	Concept Exploration	Demonstration and Validation	Full Scale Development	Production & Deployment
System Form	System Concept Paper (SCP)	Decision Coordinating Paper/ Integrated Program Summary (DCP/IPS)	System Design and Engineering Prototype	Production Quality System
Information Available	Acquisition Strategy Concepts Goals & Thresholds	Concept Alternatives Goals & Thresholds Life Cycle Management Plans	Requirements Specifications Test Plans & Results	Operational Experience
Requirements Model	Mission	System	Components	Mission/Logistics
Evaluation Scope	Mission Goals	Technical Approach	Technical Characteristics	Operational Characteristics
Basis for Evaluation	Proven Methods vs Un--Proven Methods	Complexity Constraints	Completeness of Testing	Operational Testing in the Presence of Unpredictable Stimuli
Recommendations	Determine Realistic Thresholds	Incorporate New Insights and Previous Experience	Design and Execute Progressive and Comprehensive Test Program	Assign Realistic Tolerances to Thresholds

FIGURE 3.3–2: THE SOFTWARE MATURITY MATRIX

The rows of the Software Maturity Matrix represent differing perspectives for observing the evolutionary nature of software within a system. The columns represent different phases in time corresponding to the sequence of events visible to acquisition decision makers. Thus, by reading a single row, a decision maker can determine whether or not the system is progressing along the expected path. Similarly, examination of a single column can reveal whether or not the system has reached the expected level of maturity for the current phase. Descriptions of each of the components of the matrix follow.

Acquisition Milestones

DoD 5000.3-M-1 defines a milestone to be "a major management decision point in the overall acquisition process of a major DoD system requiring Office of the Secretary of Defense (OSD) and/or DoD Component program review". These decisions include Milestone 0 which authorizes program initiation, Milestone I which selects alternative system concepts for further investigation, Milestone II which represents an intention to deploy the system selected for development, and Milestone III which is the production decision.

Acquisition Phases

The total acquisition and deployment cycle is divided into four acquisition phases:

The Concept Exploration Phase is used to identify and examine various system and development concepts which will satisfy the operational mission requirements, including the role of software within the systems.

During the Demonstration and Validation Phase, system requirements are validated and the suitability of the system for engineering development is demonstrated. In addition, alternate approaches for allocating system requirements to hardware and software are investigated.

Full Scale Development Phase activities include designing, implementing, testing and integrating the hardware and software components into a total system.

Finally, the Production and Deployment Phase results in a specified number of systems being placed in field use and maintained until retirement.

System Form/Information Available

At any point in time, the system exists in some conceptual or physical form. The available information varies as time presents more opportunities for analyzing and modifying the system concept.

During the Concept Exploration Phase, the system is embodied in the System Concept Paper (SCP) which describes the acquisition strategy, including the identification of concepts to be carried into the Demonstration and Validation Phase, and reasons for elimination of other concepts; and establishes thresholds to be met and reviewed at the next milestone.

During the Demonstration and Validation Phase, the system exists in the Decision Coordinating Paper/Integrated Program Summary (DCP/IPS) which supplements system concept alternatives and thresholds with planning for system life cycle management. Although descriptions of alternative concepts in the SCP of the Concept Exploration Phase may include information related to each concept's utilization of computer resources, the IPS is the first document that specifically requires the presentation of system computer resource issues.

During the Full Scale Development Phase, the form of the system evolves from that of system requirements and specifications into detailed hardware and software designs which result in the production of an engineering prototype. The engineering prototype, as well as each of its components, is subjected to testing that is sufficient to justify a production decision.

The system form during the Production and Deployment Phase is that of a production quality system supplemented with information concerning operational experience.

Requirements Model

This row of the matrix summarizes the progression of the system through the acquisition cycle. The system requirements initially flow from mission objectives during the Concept Exploration Phase. Once a single concept has been selected during the Demonstration and Validation Phase, the requirements for the system to be developed can be specified. Full Scale Development encompasses the refinement of the system requirements to a level of detail that allows the development and integration of each individual component. The use of the system during Production and Deployment completes the cycle by evaluating the system in an operational environment with respect to the mission goals and logistics strategy.

Evaluation Scope/Basis for Evaluation

The primary aspects of the system which are examined for the purpose of evaluation also vary with time.

During the Concept Exploration Phase, the considerations by necessity center on mission objectives and the accurate representation of the threat to be countered. If similar systems have been developed, evaluations can be made with respect to methods proven through operational deployment.

During the Demonstration and Validation Phase, evaluations are expanded to consider the planned technical approach with respect to complexities introduced into the system by known constraints. For real-time systems, these complexities will very likely include the impact of timing constraints on the software portions of the system.

The Full Scale Development Phase results in an engineering prototype whose technical characteristics can be thoroughly tested and evaluated. In addition, operational characteristics are tested to the maximum extent possible. The software test program must be founded upon a systematic, quantitative, and objective approach which is employed in a progressive manner. This ensures that information gained from early low-level tests is factored into the design of higher-level tests, eventually building to a software system evaluation prior to the testing of the completely integrated engineering prototype of the system. Confidence in evaluation results can be expressed in terms of the extent and rigor of testing.

The Production and Deployment Phase provides additional insight into operational characteristics regarding previously untestable interactions among system and software components and the operational environment.

Recommendations

Guiding principles for each of the acquisition phases follow.

Efforts during the Concept Exploration Phase must be concentrated on the determination of realistic thresholds for required operational and technical characteristics based upon available mission needs information.

During the Demonstration and Validation Phase, thresholds must be updated based upon insight gained during the validation of the requirements and experience with similar systems. Thresholds must also be specified for software characteristics based upon previously established limits for system operational and technical characteristics.

The Full Scale Development Phase must incorporate a progressive and comprehensive software test program that substantiates the achievement of specified thresholds for all required characteristics.

The Production and Deployment Phase offers the first opportunity to evaluate the system's requirements and performance in a true operational environment. Requirements specified during the development of a system are predictions of an operational need. These predictions must be updated based upon real need as observed during system deployment. At this point, tolerances can be assigned to the thresholds, thereby providing a realistic basis for the maintenance, modification, or replacement of the system.

The Software Maturity Matrix can be used to assess system and software information that is reported throughout the acquisition process. For example, suppose that at the Milestone I decision point it is reported that the software for the system has a reliability of .98. According to the matrix, in general, only information related to mission objectives is available at this point in the process. Although reliability thresholds may have been assigned, it is unlikely that the role of the software in the system has been thoroughly investigated. It is even less likely that any software has been developed and tested such that an achieved reliability figure could be reported. On the other hand, suppose that evidence revealing the expected operational reliability of the software is not available at the Milestone III decision point. If the software test program was properly structured and executed, this information would be available. In this case, a thorough review of the software test program may be in order. By scanning the column of the matrix that corresponds to the relevant time in a program's acquisition, the context within which to judge software knowledge and status is easily inferred.

A final point needs to be made concerning thresholds. The definition provided by DoD 5000.3-M-1 specifically associates thresholds with time. It embodies the concept that thresholds change as the system development progresses. These changes are based on the availability of additional information about mission needs and the technical feasibility of specified requirements. As the system evolves, the partitioning of requirements into functional capabilities and the allocation of responsibilities to hardware and software may advance through several iterations. Every system development activity results in information that must be reflected in the current thresholds for required operational and technical characteristics. This incorporation of new knowledge must be institutionalized in each program's development strategy.

CHAPTER 4

MANAGEMENT AND SCHEDULES

Effective T&E requires a well-managed flow of information. One component of this flow is vertical and connects developers, users, and acquisition decision makers. Another component is horizontal and connects issues and organizations that may be concerned with different aspects of the system. This chapter outlines an approach to managing both types of information flow. The important features of this approach include recognizing the key participants in the software T&E process and the roles that they have been given (e.g., through policy via DoDD 5000.3), and assigning spheres of responsibility to each of the participants to account for the major technical and programmatic risk elements of the system. An essential aspect of such an assignment is its focus on developing a realistic management approach that makes organizational participants responsible for T&E issues that they can control. Another essential aspect of this approach to software T&E is the maintenance of independence in the program, that is, utilizing the efforts of DT, OT, Quality Assurance (QA), and Independent Verification and Validation (IV&V) to complement each other while achieving their respective goals.

4.1 Identifying Software T&E Organizations

Software T&E planning activities begin at the initiation of system acquisition. Critical to this planning is the early identification of all program participants and the definition of their distinct roles and responsibilities. For example, OSD and Service Headquarters may place requirements on the testing process. Test activities are primarily performed by the Program Office, associated contractors, and the Service T&E Agencies. In addition, other organizations (e.g., Support Agencies and User Commands) may provide support during the definition and execution of specialized tests. Because of their varied involvements and the needs of individual programs, each group will organize differently for testing. Table 4.1-1 lists typical testing responsibilities of organizations directly involved in system acquisitions. In all cases, these organizations participate in the testing of the system, and are either directly or indirectly involved in the testing of the software components of the system.

Organization	Responsibility
Deputy Under Secretary of Defense (T&E) (DUSD(T&E))	Setting of DT&E policy within DoD, the review of TEMPs, & the provision of technical assessments to the JRMB.
Director, Operational T&E (DOT&E)	Oversight of OT&E within DoD, the review of TEMPs, & the provision of OT&E assessments to the JRMB, Secretary of Defense, & Congress.
Service Headquarters	Review of summary test results for funding, schedule, & fielding recommendations.
Development Commands	Review of summary test results for funding, schedule, & performance decisions.
Program Offices	Overall planning & sometimes conduct of the DT program. Review & approval of contractor test documents for adherence to specifications & the contract. Support of OT.
Contractors	Preparation, execution, reporting, & analysis of results of DT.
Development Test Agencies	Planning, conduct, & reporting on DT with respect to satisfying the required technical performance specifications & objectives.
Operational Test Agencies	Planning, conduct, & reporting on all OT&E with respect to system operational effectiveness & suitability. Monitoring, participation in, & review of the results of DT&E to obtain information applicable to OT&E objectives.
Software Support Agencies	In some cases, IV&V of the software & evaluation of the software for maintainability.
User and Training Commands	In some cases, support of OT & evaluations of software usability.

Table 4.1-1: Organizational Test and Evaluation Responsibilities

Once the roles and responsibilities are defined, the emphasis of the organizing activity shifts to that of determining the appropriate amount of independence that shall exist between the participants for the duration of the acquisition. The benefits derived from organizational independence include alternate interpretations of users' needs and attainable solutions, and unbiased portrayals of acquisition risk. The achievement of organizational independence requires additional manpower and resources. Decisions concerning the extent of independence appropriate for a given program must balance these costs and benefits. This basic tenet of organizing for testing, independence from development, is also important for software testing.

DoD-STD-2167 requires that high-level software tests be planned, conducted, and analyzed by a group that is independent of the software developers. This approach has been found to be most successful on past programs [STE 86]. The degree of independence strived for during lower-level tests must be decided for each program based on the risks associated with various software subsystems and components. Common approaches for accommodating independence within contractor organizations include establishing an independent software test group within the software engineering organization; assigning responsibility for high-level software test to the system engineering organization; establishing a software test group within an existing independent T&E organization; or assigning responsibility for software test to an independent QA organization.

The Program Office must allocate staff sufficient for the detailed review and approval of contractor software development and test activities. If objective decisions are to be made and contracts are to be managed intelligently, this staff must have a working knowledge of software and testing technology. In some instances, the DT Agencies can provide an independent assessment of the software development and test activities.

The OT Agencies are required by DoDD 5000.3 to be separate and independent from the Development and User Commands. These organizations typically test systems and not components of systems (including software components). However, they need to be aware of the software's intended contribution to the operational characteristics of the system and the software test results to date.

Finally, an IV&V Agency can provide detailed technical analyses of software implementations and methodologies. IV&V techniques are usually applied to augment other testing activities in areas of high risk. This role may be taken on by a contractor (other than the development contractor), the SSA, or the DT or OT Agencies. A benefit of having the SSA perform the IV&V of the software is that it allows early preparation for the eventual maintenance of the software. The SSA can also influence the software design and development to improve its maintainability.

In addition to independence, the early identification of a software manager within both the Program Office and the contractor's development team has been found to be important to the success of software development and testing [STE 86]. The software manager acts as the focal point on all software related matters and has total responsibility for the software development. In some cases, independent software test managers are also designated. Past experience indicates that the establishment of a separate software manager and associated staff in the Program Office should occur at the initiation of the program, or at least prior to any major development contract awards, to ensure proper focus on software and software test planning [STE 86].

Finally, the use of individuals in participating organizations that are experienced in similar applications, as well as software development, has been found to improve the overall quality of the software product [STE 86]. Although recent college graduates may be better versed in modern software technology than existing program personnel, the knowledge of application-specific problems and opportunities for exploitation of the technology may be lacking. In particular, software testers with both software and applications experience possess greater insight into the attributes that combine to form exemplary software test procedures and results. Furthermore, the maturity gained from involvement with similar systems can ease the burden of the tester/operator training required to ensure test realism. The proportion of high-level experienced individuals assigned to a program must be appropriate to the expected complexities.

4.2 Balancing Test and Evaluation Activities

Ultimately, decisions to employ specific types of software testing techniques must consider both the cost of the techniques and the cost of the system failures that they are designed to prevent. If a system is man-rated and performs a function directly affecting the defense of the United States, the cost of encountering failures is naturally higher than that associated with office automation system failures. Obviously, a program should be willing to apply more resources to the testing of man-rated systems than office automation systems.

Chapter 4: Management and Schedules

The goal of software testing is to demonstrate capabilities and reveal errors that may exist. The cost of an error remaining undiscovered until the software is in operational use is composed of the cost of the system failing to meet its mission objectives and the cost of repairing the error.

Historically, it has been shown that the cost of repairing software errors rises dramatically as the development progresses. Therefore, the most useful testing strategies or methodologies emphasize achievement of test objectives as soon as necessary software components are available. This does not, however, obviate the need for OT. True system behavior can only be ascertained during system employment in an operational environment using typical operator personnel.

Although all types of software testing techniques share the common goal of preventing system failures, each (e.g., DT, OT, QA, IV&V) has its own distinct objective and set of activities:

The objective of DT is to verify the attainment of technical performance specifications and supportability. DT activities include the planning, conduct, and evaluation of tests at the unit, integration, and software system levels. Unit tests are concerned with exercising the software and demonstrating the capabilities of individual components (or units). Integration testing is performed on aggregates of units that have successfully completed their unit level testing and usually emphasizes interface testing and the composition of system functions. The DT process culminates in acceptance testing of the total software system (followed by the total system) with respect to defined requirements. Testers should be involved during early program phases in the analysis of software requirements to ensure that they are testable and traceable to individual test cases.

The objective of OT is to determine the effectiveness and suitability of the system for use in combat by typical military users. Like DT, principal OT activities are the planning, conduct, and evaluation of tests. OT differs from DT in its concentration on mission needs and the functioning of the system in a typical operational environment. Again, as for DT, OT will benefit from having testers involved in the analysis of software requirements to ensure testability and traceabilty.

The objective of QA is to ensure adherence to standards, conventions, and requirements, and successful completion of activities according to specified criteria. QA activities include defining software development standards and procedures, and monitoring the software process and products to ensure compliance.

The objective of IV&V is to ensure that the software will not fail in its operational mission, either by failing to correctly perform an intended function or by unintentionally performing an undesirable function. Verification activities examine the product of each phase of software development (e.g., requirements analysis, preliminary design, detailed design, coding) for consistency, completeness, and traceability to the product of the previous phase. Validation activities demonstrate the consistency, completeness, and traceability of the final software product to its original requirements.

Table 4.2-1 indicates which of the above types of testing techniques would normally be expected to contribute to the evaluation of the software's achievement of thresholds with respect to the operational and technical characteristics discussed in Section 3.1. It should be noted that the omission of an X in a cell of this figure does not indicate the prohibition of a type of testing technique for the investigation of a characteristic. In some cases, it may be desirable to supplement the activities described with additional analyses.

Operational effectiveness characteristics are best demonstrated through the execution of the software. Although the true behavior of the software can only be demonstrated in the actual operational environment, the conduct of OT, and the correction of errors found during OT, are too expensive to incur without some confidence that the software will achieve the required thresholds. Once this confidence is obtained through DT, OT is employed to substantiate the DT results. In addition, for mission critical software or high risk software components, DT and OT may be supplemented with examination by an IV&V Agency.

It is also important to demonstrate the operational suitability characteristics through OT. With the exception of maintainability, all of these characteristics can also be examined by DT, and should be for the reasons noted above. In instances where stringent requirements have been placed on the characteristics, IV&V can be applied to reduce the associated risks. For maintainability, a dynamic test would consist of making a specified change to the software while measuring attributes of interest (e.g., time, cost). The reason that DT or the application of IV&V techniques are not advocated for maintainability evaluations is that the information gained from dynamic tests in a laboratory setting cannot be used to infer that the software will be maintainable by the actual SSA utilizing the true support environment.

	DT	OT	QA	IV&V
Operational Effectiveness Characteristics	X	X		X
Operational Suitability Characteristics				
Availability	X	X		X
Integrity	X	X	X	X
Interoperability	X	X	X	X
Maintainability		X	X	
Reliability	X	X		X
Usability	X	X	X	
Technical Characteristics				
Correctness	X			X
Efficiency	X			X
Expandability	X		X	
Flexibility			X	
Portability	X		X	
Reusability			X	
Testability	X			X

Table 4.2-1: Scope of Test Activities

When attainment of an operational suitability characteristic is determined, in part, by adherence to selected standards or conventions, QA techniques can be used to support the required evaluation in a cost effective manner. For instance, programming and documentation standards are usually required to support maintainability goals. Examples of other standards and conventions associated with operational suitability characteristics include communications protocols that promote interoperability, accepted keystroke sequences that may enhance usability, and standard encryption techniques that provide integrity. The use of QA techniques to examine the implementation of required standards would be expected prior to the conduct of OT.

Whereas the demonstration of the operational characteristics discussed above ultimately depended upon OT, the demonstration of technical software characteristics depends upon the results of DT to the exclusion of all OT. Recall that in Section 3.2 the process of refining operational characteristics to low level requirements eventually uncovered technical characteristics. As the system is constructed, the technical characteristics must be demonstrated as a foundation upon which to achieve the required operational characteristics.

In this case, DT is supplemented by either QA or IV&V activities. Once again, QA is employed in instances where the use of standards or conventions contributes to the achievement of technical goals. IV&V techniques tend to be more appropriate when dynamic testing is the primary method of demonstration. In Table 4.2-1, only QA activities are associated with the technical characteristics of flexibility and reusability. This is because it is believed that the use of modern programming practices enhances the ability to achieve these characteristics which have gained interest very recently and are not yet well defined.

In the discussions above, IV&V activities were recommended to supplement DT and OT when the software was mission critical or introduced substantial risk to the acquisition. It should be realized that there is a broad spectrum of IV&V activities which can be used on a specific program [Orl 84]. For instance, if a minimal IV&V program is desired, its activities should concentrate on the establishment of a good requirements baseline and development procedures. In addition, a thorough analysis of the test program should be conducted. If a maximal IV&V program is desired, its activities should establish a high level of confidence in all aspects of the system including the design and implementation. This would include independent testing as opposed to independent analysis of the DT program.

In addition to DT, OT, QA, and IV&V activities, specialized analyses may be performed with respect to selected operational and technical characteristics. For example, availability or reliability models may be used to predict achievement of characteristic thresholds. These models must be validated. When usability presents critical issues, human factors experiments may be conducted. When security is of prime concern, special certifications must be obtained. These analyses compliment those associated with DT, OT, QA, and IV&V.

There are many opportunities for overlap between DT, OT, QA, and IV&V activities. When striving to minimize redundancy, those activities that can be accomplished earliest in the development process should be chosen and assigned to the organizations which can provide the desired independence as described in Section 4.1. For example, if a critical operational software issue can be addressed during DT as opposed to OT, the OT Agency should ensure that DT plans and procedures will produce the necessary data for independent evaluation by OT personnel.

4.3 Sharing Information between Organizations

To effectively utilize the limited resources available for testing, information must be shared among the organizations involved in the test process. Two types of information are of interest: test planning information and test results. Test planning information is evaluated to determine the opportunities for combined testing or elimination of redundant tests. Test results are assessed to determine progress and the impact of less than satisfactory results on other planned tests.

The acquisition decision-making hierarchy is organized around a vertical flow of information. DoDD 5000.3 requires a software T&E program that provides for "effective sharing of test results across life cycle phases as well as improved vertical flow of information within the decision-making structures..." In general, test results are summarized as they are passed from the lower to the higher decision levels of a program. In other words, producers and consumers of information operate at different levels of abstraction. To achieve effective sharing of information, the consumers need to be able to access information at a level appropriate for their review, and data at each level of abstraction must be maintained for future reference. Two requirements need to be satisfied to accomplish information sharing:

First, early in the structuring of the program, agreements must be reached that allow participating organizations access to software test information, as needed. These agreements must accommodate consumer needs for access to uncompressed information so that independent analyses of test results may be accomplished. For example, it may be appropriate for an operational tester to access the results of DT but the operational tester must perform his own analysis and draw his own conclusions. In addition, special requirements for data collection must be concurred with and documented. The documents may take the form of memorandums of agreement between government agencies or contracts between the program office and industry participants.

Second, physical communications vehicles are needed to allow the information to be propagated, in a timely manner. An ideal model of this would be a network of providers and consumers with access to all information at any level of abstraction. Since this model is not likely to be implemented on most programs, the users should establish requirements for the least amount of information necessary to conduct their activities, while reserving the right to access more detailed information if the need arises.

A special case of information sharing arises when an IV&V effort is utilized. The IV&V team must receive current information throughout the software development to technically evaluate the process and products of the development team. The IV&V team must also report results of its analyses in a manner that allows the development effort to benefit from the findings. Special timing considerations arise when relevant information must be transmitted via the program office to maintain the independence of the parallel efforts of the development and IV&V teams. Time is required for the program office to examine the development team's products and the IV&V reports before relaying all necessary and appropriate information to each respective team. It is imperative that arrangements be made very early in the program life cycle for the timely and efficient sharing of information between these groups.

If agreements are not reached early in the acquisition process concerning the information to be shared among organizations and the responsibilities of each organization with respect to collecting, processing, and reporting results, there is a risk that opportunities for data gathering will be overlooked. In some cases, the only way to recapture the lost data may be to completely recreate the missed test.

4.4 Scheduling Software-Related Events

When constructing software development and test schedules, key software items must be available to support software T&E events that culminate in key software subsystem demonstrations. These T&E events must be scheduled to support system level tests and demonstrations. The scheduling of these basic events must also accommodate integration, analysis, and regression testing requirements, and take into consideration the availability of personnel, testing tools, and other support items.

It is interesting to note that, in fact, the entire software development schedule is driven by the system/software test schedules and therefore should be derived from them. When a new program is authorized, the date of the availability of its Initial Operational Capability (IOC) is specified. Following this, the major events that culminate in a working system are scheduled to coincide with the specified IOC. These major events include test events (e.g., key subsystem demonstrations). In order for testing to occur on subsystems, for example, the subsystem's components must have been previously integrated. Prior to integration, unit testing must occur, and prior to unit testing the units must have been developed. Thus, the development schedule is derived from the test schedule. Each event of the overall schedule is planned to achieve the goal IOC. From this, it should be obvious that a key element of successful programs is the initiation of test planning activities very early in the development schedule.

The overall program schedule must allow time for the integration of software components, as well as for the integration of software and hardware components. Even though individual software components may have satisfied their unit testing requirements, time must be allotted for integrating the components and getting the aggregate to operate satisfactorily prior to the initiation of integration testing. Likewise, time must be allocated for the integration of software and hardware components prior to scheduled system integration tests.

Another activity that consumes time and should be explicitly allowed for in test schedules is the analysis of software test results. The amount of time allotted must include that needed to obtain test results, which may be observed directly or received from other organizations that conducted or participated in the test. Time must also be allotted for the interpretation and analysis of the results, the determination of whether or not the actual test satisfied test plan requirements, and the production of test reports. Finally, any dependencies that exist between tests must be carefully considered so that scheduled start dates of dependent tests allow time for the required analyses of previous test results.

Regression testing is another essential part of software development. After errors have been located and repaired, regression testing is performed to ensure that no new errors have been introduced during the maintenance activity. Since it is not possible to predict how many errors will exist in each software component, it is difficult to determine a priori how much time must be allocated for regression testing in the test schedule. When possible, estimates of appropriate allocations should benefit from past experience with similar systems. Any schedule that does not provide for regression testing is suffering from a case of hopeless optimism and should be rejected.

Tools used to test software are usually software programs themselves. Thus, the overall program schedule must include milestones for the acquisition and test of the software test tools. Important considerations when developing software test schedules include:

If the necessary software test tools are not available OTS, insure that the program milestones allow for their development and test prior to their scheduled use. Appropriate margins should be incorporated into the test schedule so that any problems that may arise during the tool development will not impact the testing of the system software. Contingency plans should be made to allow the continuance of the overall test program without the newly developed tools, if needed.

Whether the tool is custom built or OTS, the overall test schedule should allow for the validation of the tool after delivery. Since evaluations of the effectiveness and suitability of the system software may depend heavily on information provided by the software testing tools, it is imperative that the tools themselves be carefully tested and evaluated prior to their use.

Since many software testing tools are large consumers of computational resources, schedules must allow either time for the acquisition of additional resources to support the tools, or adequate time to permit their use within the confines of existing resources.

Finally, the test schedule must accommodate training and familiarization of test personnel with the new tools.

In addition to software test tools, the availability of other support items, such as system hardware, simulators, or special test equipment, may also need to be considered when scheduling test events. The use of these items depends not only on their availability, but also on their correct performance and the availability of any personnel needed for their operation. In some instances, schedules must also provide dedicated system time to allow potentially destructive software tests to occur.

Finally, the scheduling of software tests must take into consideration the availability of personnel that will be necessary to conduct the tests. For example, technicians, hardware designers and testers, software designers and testers, and computer operators are some of the personnel that may be required to conduct a particular software test. Furthermore, schedules must allow for training personnel with respect to the system requirements, test requirements, and operation of the new system.

4.5 Utilizing Technology as a Management Aid

The planning and management of software testing, described in this chapter, can be aided by the application of appropriate technology. Automation should be used wherever possible to enhance the decision making capabilities of managers. The use of a computer network to share test information is an example of modern technology benefiting the decision making process by allowing access to information that would not otherwise be available for consideration in the required time frame. OTS cost estimation and tracking systems, spread sheet packages, and scheduling and monitoring tools, in many instances, can be used to increase the effectiveness of management. Automated decision support systems with accompanying databases and communications capabilities are also available commercially. Program management personnel should be aware of available technology and perform cost-benefit analyses to select management aids for application to their program.

CHAPTER 5

PLANNING AND REPORTING

Careful and realistic test planning is the key to successful test programs. By the same token, test reports document the execution of a test plan and are therefore critical to the assessments made as a result of T&E. A common factor in many troublesome acquisition programs is the lack of detailed and effective test plans and the absence of adequate mechanisms for reporting test results. On the other hand, early formulation of test plans and timely reporting of results, allow designers, managers, and executive decision makers to identify technical and administrative deviations from program plans and to respond quickly, often-times saving the program from expensive corrective measures later on.

Many institutional forces work against detailed software test planning. Frequent objections include claims that test plans and reporting requirements increase development costs and invite micro-management of the development effort. A common excuse for delaying or ignoring software test plans is that critical software properties cannot be specified to any useful degree until late in the acquisition process. These represent legitimate concerns, but a well-structured test program can address these issues while still providing for early and detailed planning aimed at determining the status of the software.

In simple economic terms alone, early and complete test planning aimed at assessing software quality, freedom from design defect, and overall capability, can repay even large investments many times over. Figure 5.0-1 shows one way in which early testing efforts can help reduce overall program costs [Boe 76]. The chart illustrates the incremental (relative) cost of finding and removing software errors during the various software life cycle phases. This study indicated that the increase in the cost of finding and removing errors as the software moves to operation and maintenance can be as much as 100 times the corresponding costs during early life cycle stages. The costs due to testing would have to be extraordinarily large to offset not only the defect removal costs, but also the operational costs stemming from logistics delays, lost system functions, and possible mission failure. Even in projects with minimal attention to testing, as much as 40% of the total software life cycle costs can represent testing. The only way to ensure that this investment is effectively utilized is through test planning and reporting.

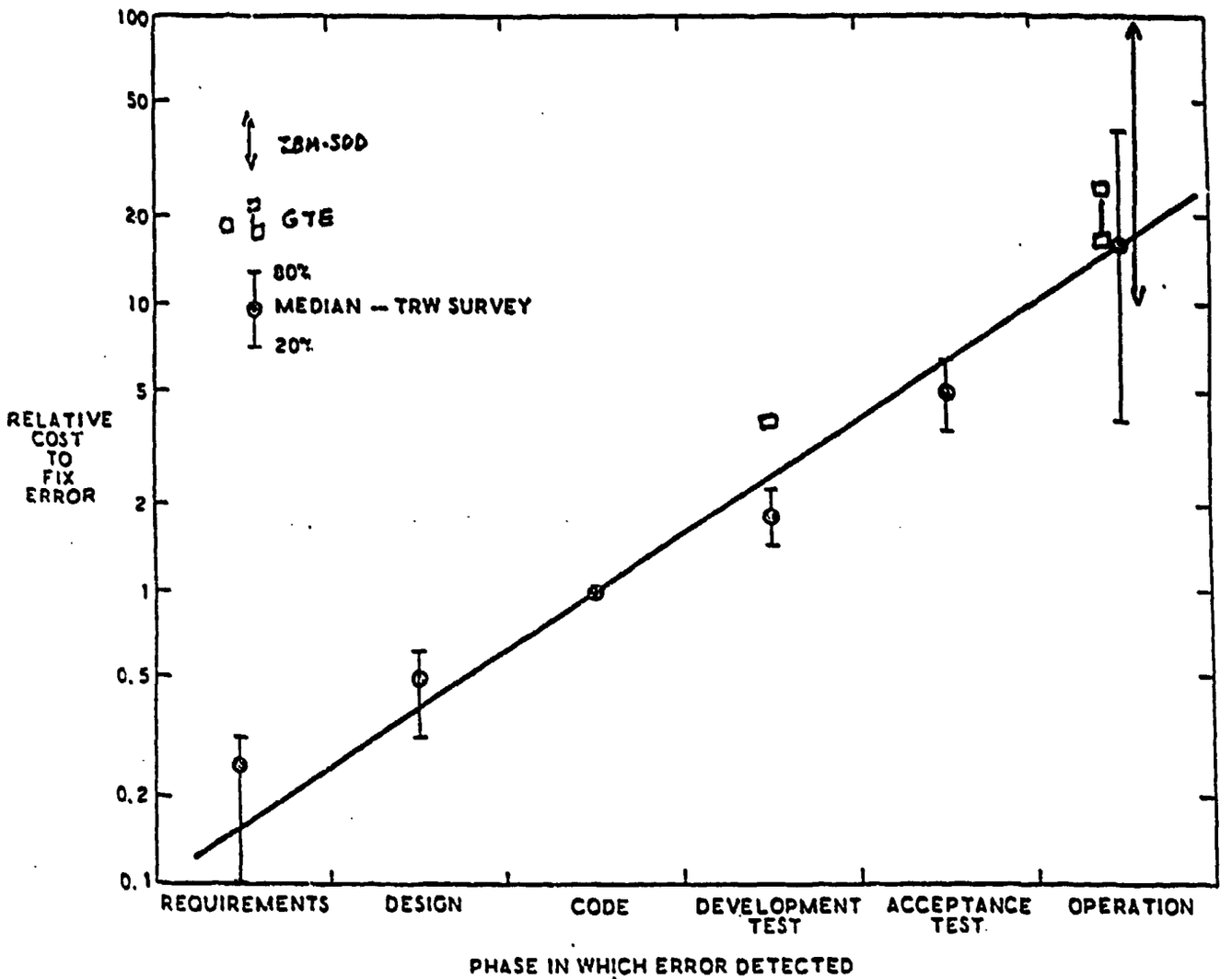

FIGURE 5.0–1: RELATIVE COST OF ERROR CORRECTION

Chapter 5: Planning and Reporting

5.1 Understanding the Test Planning Process

Establishing a test program that requires rigorously adhered to software test plans and detailed test reports may be resisted by technical managers. For one thing, test plans represent a chain of accountability. In particular, each management and technical level in the programmatic structure of a development program is responsible for a link in a chain of test plans. This gives managers increased visibility into the status of the software and requires the specification of success criteria for the project as a whole. In poorly managed programs, this level of visibility may indeed be used to micro-manage or otherwise subvert the program organization. In well managed programs, however, the test plans fit into the project organization. As will be described below, the planning process can be integrated with the overall engineering effort. It does not by itself cause programs to either succeed or fail. However, an effective test planning process is a powerful instrument that is more often associated with good program management and technical success than with unsuccessful programs.

Each acquisition program must eventually answer the question of how to best include software-specific issues in test plans. In previous chapters, the problems of identifying software risks, requirements, and test issues have been treated in some detail. The test planning process described in this chapter is a logical next step: once the software-specific critical T&E issues have been identified, tests are planned whose objectives include the resolution of outstanding T&E issues.

The beginning of the chain of test plans is the TEMP required by DoDD 5000.3. This plan is a system level document that defines and integrates system characteristics, critical issues, test objectives, responsibilities, resources, and schedules for T&E. As a matter of policy, the TEMP contains descriptions of the planned and to-date software T&E effort for all software components that implement mission critical functions or represent special sources of risk in the system acquisition.

Below the levels addressed by the TEMP are test plans for the increasingly detailed treatment of software. These levels include tests of software subsystems and configuration items, tests that reflect the status of integrating software components and functions with each other and with hardware components, and tests that are applied to the components as well as to the lower level structures or units from which they are composed. At each of these levels, more detailed plans elaborate the plans of the levels above them in the hierarchy. Results of these tests combine to summarize the extent to which more general T&E issues have been resolved by the test. This process is the chief vehicle for the "vertical flow of information" required by DoDD 5000.3.

The development, acquisition, and support of MCCR software is governed by Service regulations for the management of computer resources in defense systems. In addition, DoD-STD-2167 provides requirements for inclusion in contracts for the development and acquisition of MCCR software. DoD-STD-2167 is a product of initiatives sponsored by the Joint Logistics Commanders and is for use by all Military Services.

As described in Section 2.2, DoD-STD-2167 assumes the existence of a SSS and draft SRS as a starting point for the software development process. Although the SSS and SRS are primarily requirements documents, each includes a section that defines qualification requirements and therefore contributes to the test documentation chain. The qualification requirements section of the SSS contains the testing philosophy and overall approach to be followed, the assignment of responsibilities for test performance, requirements and constraints on formal testing, and a cross reference between system requirements and qualification methods, levels, and formal test requirements. The qualification requirements section of the SRS specifies the methods, techniques, tools, and acceptance tolerance limits necessary to establish satisfactory software quality. If qualification and testing requirements are to be adequately reflected in the SRS, software test planning must commence in concert with the specification of the software requirements.

When executing the DoD-STD-2167 test process, information is created in the form of Software Test Plans (STPs), Software Test Descriptions (STDs), Software Test Procedures (STPRs), and Software Test Reports (STRs). Amplifying the SRS test-related requirements, the STPs set requirements, outline organizations and responsibilities, specify resources required, and provide testing schedules. The STDs include input data, expected output data, and evaluation criteria. The STPRs contain the step by step details for test conduct. Finally, the STRs summarize the test, results, and analysis, and provide recommendations.

The detailed formats of individual test plans are specified by applicable policy, regulations, or standards. However, all test planning documents should contain the following information.

System Description: a description and identification of the system, subsystem, or components that are to be tested. Included in the system description is a brief discussion of any operational concepts that may affect an evaluation of the test results.

Test Objectives: a summary of the technical or operational characteristics to be demonstrated by the test. Most important to articulating these objectives are the required or threshold values for the characteristics and the specific questions or issues that must be answered in order to determine whether or not a threshold value is achieved.

Past Testing Summary: an evaluation of past test results that includes a description of objectives achieved and an indication of issues left unresolved by previous tests.

Planned Tests: specifications of tests or methodologies to be applied. These should be in sufficient detail to indicate that the test objectives will be satisfied.

Programmatic Summaries: information pertaining to test budgets, schedules, resources, and organizational matters.

At the level of the TEMP, these planning elements deal in very specific terms with the relationship between the software and other system components, particularly insofar as the software's status will affect assessments of overall system worth. In more detailed test plans, such as those required by DoD-STD-2167, the issues will have been refined to reflect more specific aspects of the software's design and construction. At the still more detailed levels that correspond to test procedures, the test "methodologies" may, in fact, be detailed descriptions of test sequences and the expected results that the tester will use during the conduct of the test.

Regardless of the level of test plan, all of the elements noted above are essential aspects of the planning documents. By requiring the test organization to state its objectives in conducting the test, to give explicit parameters by which the system's success or failure is to be judged, and to explain the manner in which the test supports these, the program is protected from ad hoc and ineffective testing approaches. A plan structured in this way also serves as a "contract" that binds evaluators and developers to observe the same evaluation and success criteria, so that even over very long periods of time, consistent management controls can be exercised over the development effort. Another key aspect of plans organized in this manner is their ability to evolve. Each such plan is a "living document" that, in addition to presenting testing which has not yet been carried out, contains the history of the development effort as revealed by the testing that has been conducted at that level. For example, the TEMP is a living document that is updated annually and addresses the changing critical issues affecting a major acquisition. It is essential that test plans reflect the actual program progress.

Finally, the importance of the creation and maintenance of the complete test documentation chain for a program cannot be overemphasized. In some cases, future events may result in questions that can be answered by analyzing existing test results, possibly supplemented with results from limited new testing. If evidence concerning software behavior and quality is improperly recorded or lost, a program cannot benefit from such economies. In fact, a lack of proper documentation may call into question the sufficiency of the total test program.

The following sections of this chapter provide a roadmap for constructing effective test plans at all of these levels. Like all roadmaps, there is a certain amount of detail that cannot be supplied here. However, there are five major pitfalls that, if left undetected, can subvert even the most well-conceived test plan.

Vague Requirements: While testers do not directly define operational and technical requirements, they are responsible for deriving test objectives that address those requirements. Vague requirements that are not stated in testable forms, or that do not adequately constrain developers to design a system that meets user requirements cannot be used as a basis for effective test planning.

Inappropriate Standards and Contracts: Test resources are scarce. The resource base can be quickly depleted responding to contractual requirements and standards that do not support the test objectives for the system. The use of established standards, even if they are tailored for the contract, has been found to improve the overall quality of testing as well as software development [STE 86].

Ad Hoc Methodologies: T&E evaluations are based on assessments of capabilities as described in test reports. Use of methodologies that are scientifically unsound, not accepted for use in standard practice, or make use of unvalidated or undemonstrated techniques and models, reduces the soundness of the assessments. Successful software testing of major systems has historically been achieved by using systematic test approaches for both integration and requirements verification [STE 86].

Unsharable and Non-Repeatable Results: Without sharable and repeatable tests, there can be no vertical flow of information between testers (and evaluators). In particular, tests and evaluations of tests at one level in the hierarchy might have to recreate essential tests performed at subordinate levels. This can increase the cost of tests dramatically.

Not Connecting Software to the Rest of the System: All software tests ultimately support system-level assessments of suitability and effectiveness. If the software test program is not defined in concert with the system as a whole, the final and most important part of the chain of test plans is not complete.

In each case the source of each problem described above can be corrected if it is detected early enough. However, if the problem is not corrected, it can cause major difficulties, not only for the test program, but also for the development and acquisition programs at large.

5.2 Planning for the Demonstration of Software Characteristics

The general DoDD 5000.3 provisions for software testing include the following:

a. Testing to ensure that system and mission objectives will not be impaired by improperly designed, implemented, or maintained software.

b. Articulation of quantitative goals and thresholds for the required technical and operational characteristics of software components and subsystems responsible for carrying out critical mission functions.

c. Testing of software to achieve a balanced risk with the hardware.

d. Use of systematic, quantitative and objectively reportable software tests to ensure that subsequent evaluations represent the status of the software in the most realistic terms possible.

e. Institutionalization of a progressive approach to software testing to provide for effective sharing of test results across life cycle phases as well as improved vertical flow of information within the decision-making structures of the Military Service and OSD levels.

The following subsections describe the major planning activities that should be carried out to satisfy these criteria.

Relating Test Objectives to Critical T&E Issues

T&E issues are questions that must be answered in order to determine whether the system possesses a given operational and technical characteristic. Depending on the type of characteristic being addressed, the issue will be addressed in either system level or component level test plans. In many cases, operational issues are treated in system level plans and technical issues are reserved for component level plans. Frequently, however, software T&E issues center on one or more components regardless of whether or not the issue is technical. For example, a computer controlled radar system may have an operational (effectiveness) requirement to track 1,000 targets. One issue derived from this requirement may be whether or not there is a combination of inputs achievable during a given time interval that causes a software buffer to overflow.

As described in Section 3.3, critical T&E issues are derived by a decomposition of more global issues. The development of the issues reflects the design process. It is a "top-down" process in which each new issue refines the one from which it descends. Tests are designed to resolve an issue. Each test has an explicit objective that relates to a critical T&E issue. Thus, an issue that questions whether or not a software message processing system correctly processes all messages of type M1, M2, and M3 may be addressed by the following test objective:

Objective A: This test will demonstrate that all messages of type M1 that contain less than 10 binary digits (bits), any 26,000 randomly chosen messages of type M2, and all 5 possible messages that can be produced by the source of M3-type messages are processed properly.

Notice that even in this hypothetical example, the conditions of the test are spelled out in considerable detail. The problem of whether or not the three kinds of tests chosen for message types M1-M3 determine an answer to the corresponding issue is the central one to be solved in designing the test. If T&E issues are derived in "top-down" fashion, then test objectives are designed to respond to the issues in a "bottom-up", or progressive, way. In other words, objectives that address technical issues at the component level must be designed, and tests must be performed to meet those objectives, before higher level test objectives can be formulated. A frequent mistake made in even small-scale software development efforts is to ignore this hierarchical process; in particular, to wait until the software is integrated into systems or subsystems to formulate test objectives. The resulting objectives tend to be very complex. Furthermore, tracing the source of unresolved issues tends to be labor-intensive and time-consuming. On the other hand, test objectives that build upon each other allow more opportunity to treat T&E issues at the appropriate level of generality, increasing the overall manageability of the testing effort.

Suppose that, in the example above, the message processing software is actually the software driver for the function in an aircraft heads up display (HUD) processor. This processor receives signals (message type M1) from a pointing device (such as a trakball) that the pilot controls and uses to move an aiming symbol across the display. Alternatively the processor receives signals from a suite of instruments (message types M2 and M3). A higher level test objective may be:

Objective B: Demonstrate that, when stimulated with signals generated by a certain simulator, the HUD processor moves the aiming symbol to the correct point on the display.

The detailed test objective "A" supports this higher level objective. However, attempting to test to this objective, "B", directly can be very damaging to the test planning hierarchy. For example, without first resolving the more detailed issues of whether the sampling function for the pilot's pointing devices (at most 10 bits per sample cycle) has been properly implemented, any attempt to resolve objective "B" leads to tests of great complexity in which anomalies are very difficult to trace.

Even though there is an explicit relationship between test objectives and T&E issues, it may not be a direct and obvious relationship. The most common situation in which this leads to complexities in designing tests is when the critical T&E issue questions the statistical likelihood of a certain event, but the test objective (for reasons of cost or technical feasibility) does not. For example, in support of an operational requirement, it may be questioned whether or not 98% of all messages received by a software subsystem are processed with a certain outcome. In the test objective "A", this problem is resolved by dividing the messages that can be received into three types. There are only 5 distinct messages of type M3 so whether or not they can all be processed correctly can be determined by exhaustive analysis. Similarly, the physical properties of the pilot's pointing devices make it impossible for message M1 to contain more than 10 bits, so by exhaustively testing all 1,024 10 bit messages the software's complete response to M1 messages can be demonstrated.

Now, exhaustively testing M2-type messages may be infeasible. But assume these messages have been produced by a uniformly random source and that the software responds incorrectly to a given message with a known probability. After observing 26,000 message-outcome pairs and seeing at least 25,800 processed properly, it may be possible (because of the way the problem is set up mathematically) to conclude that 99% of the time all messages will be processed correctly. If that is the case, then satisfying the test objective indeed guarantees that 98% of the messages are processed correctly.

As the test objectives become more detailed and highly specific to the software being tested, statistical justifications become more abstract and less reliable. In these cases, the objectives may strive to demonstrate software capability in "typical" cases or in cases that correspond to especially sensitive "extreme" values. If the objective is to support a statistical requirement, and if the test designer has only a qualitative conception of how the typical and extreme cases relate to the quantitative requirements, then some extra margins of safety may need to be built into the test. This may involve, for example, tests whose objective is to demonstrate that an error of a certain type is not present in the software or that certain quality-enhancing measures have been taken in the software design.

No matter how thorough the test nor how careful the design, some issues will remain unresolved. Depending on their criticality, these issues may affect system level risk assessments. Sometimes the system is allowed to proceed along to the next programmatic acquisition phase even though major issues are still unresolved. This might occur because of an estimate of the expected response of the system to a known threat. It might also occur because the demonstrated test objectives indicate that the critical issues will eventually be resolved in the mature system (it is, for example, possible to initiate "reliability growth programs" that involve extensive redesign and retesting of components or even whole subsystems to resolve reliability issues). In specifying test objectives, it is helpful to know whether or not these unresolved issues are to be incorporated into new test objectives. In general, test objectives that have not been fully met are left "open" through one mechanism or another. One possibility is to maintain two or more concurrent tests, conducted by perhaps different organizations, to resolve issues at differing levels of detail. In this case, two distinct objectives may be addressed in test plans directed at entirely different groups of testers. Another possibility is to note the extent to which an issue has not been resolved and continue to the next level of the hierarchy. In this case, test objectives are really an accumulation of the objectives for the current level and the unmet objectives from lower levels. Although the management of the test program becomes more complex, it is possible to mix these two possibilities in the same test program. In all cases, however, the updated test plan (in one of the formats described above) will reflect the demonstrated status of the system as revealed by the extent to which critical T&E issues have been resolved by meeting appropriate test objectives.

Constructing a Traceability Matrix: An Example

Thus far, this manual has discussed the determination of required software operational and technical characteristics, the identification of software T&E issues, and the definition of software test objectives. The importance of traceability and the maintenance of a complete and accurate test documentation chain has been stressed throughout. Therefore, prior to describing the testing approaches that can be employed to satisfy specific software test objectives, the process of constructing a traceability matrix to aid in the control of software testing will be illustrated.

A traceability matrix is a tool that can be used to track the decomposition of system requirements from the system level to the implementation level. For each requirement below the system level, a traceability matrix documents its source in a higher level specification. It can be used to highlight instances where requirements have crept into the system at a level below the system specification and where requirements have been overlooked and not allocated to lower level components at all. In addition, traceability matrices can be used to reflect the completeness of planned testing at each level by including pointers to the test case(s) that will be used to verify each function, component, or unit. Since the elements of information that are reported in the traceability matrices become available as the software development progresses, the traceability matrices are living documents that grow and evolve in concert with the software.

The following paragraphs describe the requirements and components used to construct the sample traceability matrices depicted in Figures 5.2-1 through 5.2-4. Each entry in a matrix is an example of a partial requirement, component, or unit. Also included is an example of an associated test case that would verify a portion of the entry.

Mission/Function Matrix Requirement

Mission: Carry Out Electronic Communications.

Function: The ZZZ System shall produce target messages in a format compatible with the XXX Communications System for inclusion in the target track file of the WWW Command and Control System.

The system mission/function matrix described in Section 2.1 and illustrated in Figure 2.1-2 will be used as the starting point for this example. Suppose that the system under development is a Sensor System that interfaces with a Communications System, and that ultimately the data's destination is a Command and Control System.

System/Segment Specification Requirement

The following requirement is an excerpt from the requirements of the Target Message Reporting Function of the Communications Subsystem of the Sensor System. It has been abbreviated to simplify this example and the relevant tracing to lower levels of the system decomposition (see Figure 5.2-1).

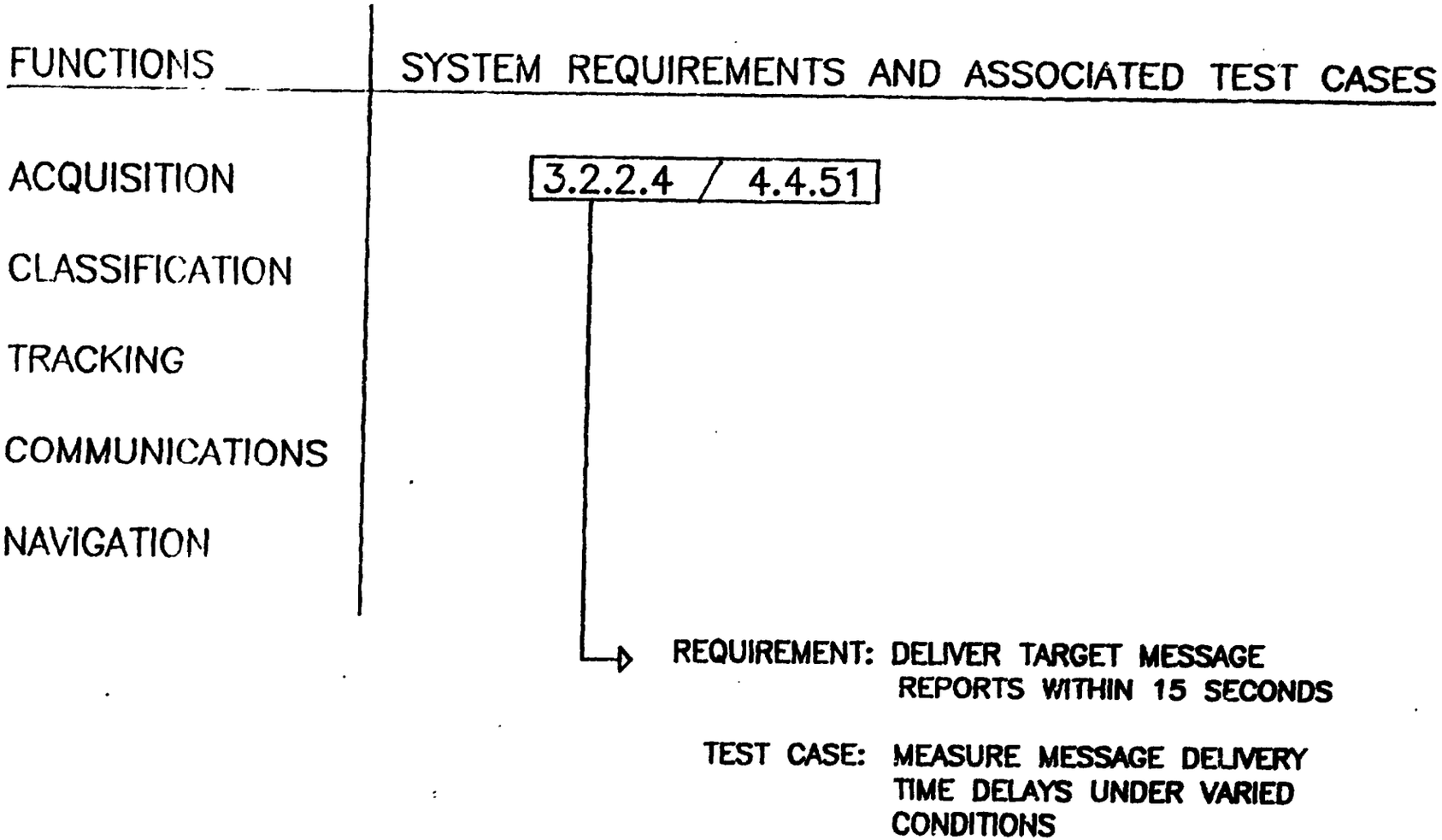

FIGURE 5.2–1: FUNCTIONS TO SYSTEM REQUIREMENTS TRACEABILITY MATRIX

3.2.2 Communications Subsystem

3.2.2.x ...

3.2.2.4 Target Message Reporting

The Communications Subsystem of the ZZZ Sensor System shall format and output messages to the XXX Communications System for delivery to the WWW Command and Control System. The Target Message Reports shall be in the QQQ format and shall be delivered without aging beyond 15 seconds of track establishment. The Communications Subsystem shall retransmit errored messages (as detected and reported by the XXX Communications System) within 1 second of notification of errored delivery...

The following test case could be used to test the Target Message Reports aging requirement. It is based on several assumed requirements of the system. First, the Target Message Reports have a time field that reflects track establishment. Second, the report delay is assumed to be a function of system load. Therefore, the test case is based on measuring throughput delay from the time of Target Message Report generation to delivery as a function of system load.

This is, of course, only an example. In a real system, items such as the following would also play an important role in the development of an appropriate test: the statistical properties of system load, other system loading requirements and priorities, other messages that may be interfering with deliveries to the Communications System, requirements for internal error recovery or fault tolerance, the fidelity of relevant simulations, the extent of available measurement facilities, and the required accuracy of the test.

4.4.51 Test Case for Target Message Reporting

Simulate or produce the system load as specified in other sections of the specification, produce targets for reporting (either through simulation or other means), and then measure the time delay from track establishment to delivery to the XXX Communications System.

Software Requirements

The SSS requirement stated above is decomposed and allocated to hardware and software components as follows: the message transmission requirements are allocated to hardware and the message formatting requirements are allocated to software. In addition, the response to the retransmission request is allocated to software. The paragraphs of the software requirements specification that reflect the allocation of message processing requirements are included below (see Figure 5.2-2).

3.2.8 Communications Processing

3.2.8.x ...

3.2.8.3 Target Message Reporting

3.2.8.3.1 Inputs

1) Established Target Track File from Tracking Function

2) Errored Transmission Indicator from Communications Processing Function

3.2.8.3.2 Processing

The Target Message Reporting Function shall maintain a list of Delivered Message Indicators in the Established Target Track File. This list shall be used to determine the age of tracks and ensure that all established tracks are reported within 15 seconds of establishment. In addition, this list shall be used to determine which messages have to be retransmitted due to Errored Transmission Indicators received from the Communications Processing Function. The Target Message Reporting Function shall construct the Target Message Reports in accordance with Mil-Std-aaaa, format QQQ.

3.2.8.3.3 Outputs

1) Target Message Reports to Communications Processing Function

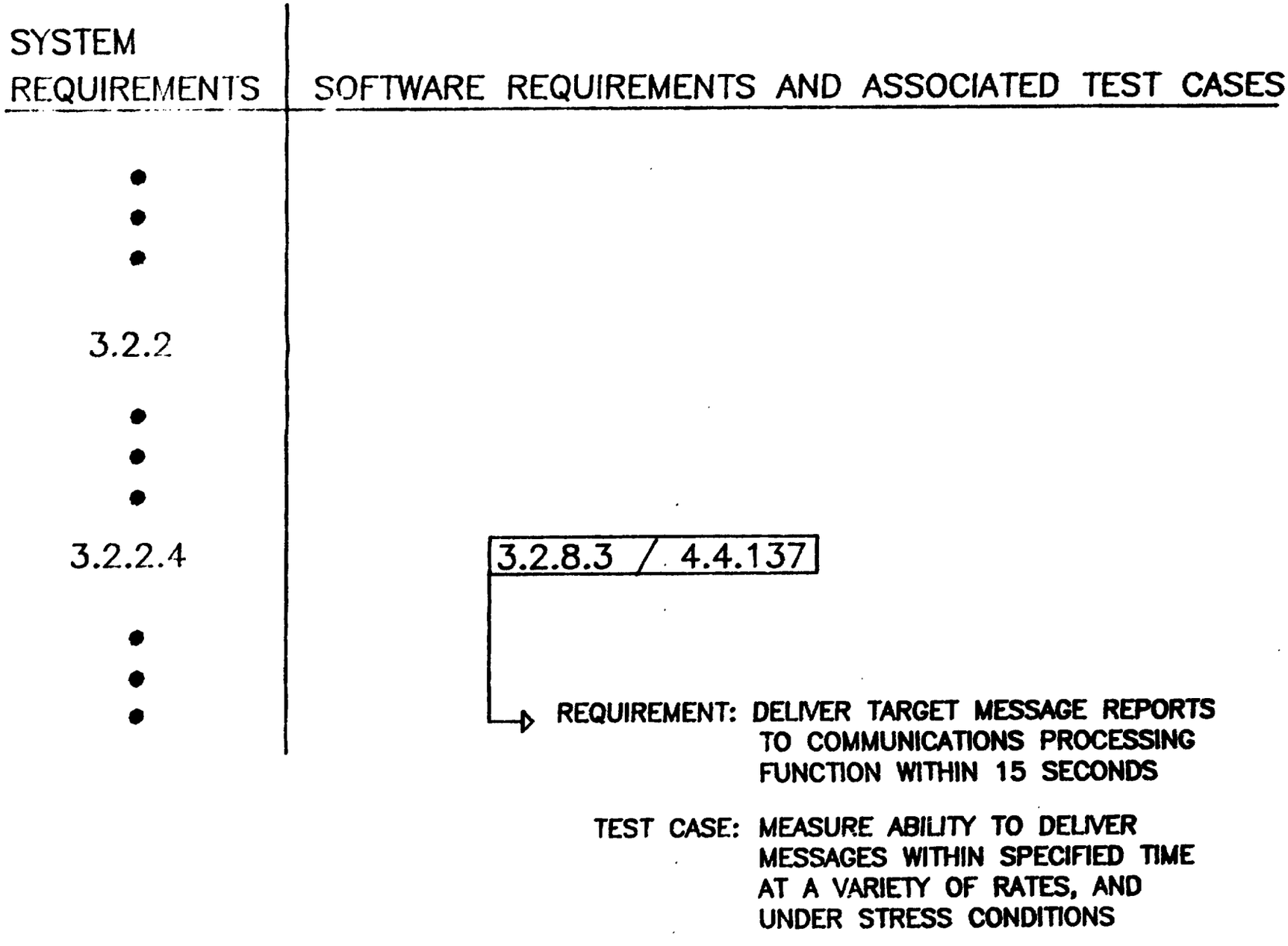

FIGURE 5.2-2: SYSTEM TO SOFTWARE REQUIREMENTS TRACEABILITY MATRIX

The requirement for augmenting the Established Target Track File implies that this function receives the Target Track File for transmission to the Communications Processing Function. In most designs, this would actually be implemented as a passing of messages, with the Tracking Function maintaining the Target Track File and just sending marks or messages to the Target Message Reporting Function telling it which messages are to be transmitted. In addition, this function would return a message or mark that indicates which messages need to be retried due to errors. The reason for pointing this out is that functional requirements don't necessarily reflect the actual design. Therefore, tests of functional requirements need to be as design independent as possible.

4.4.137 Test Case for Target Message Reporting Function

Determine the Target Message Reporting Function's ability to deliver Target Message Reports to the Communications Processing Function within its specified time, as well as its ability to react to a variety of delivery rates for Target Message Reports. This test case shall verify that Target Message Reports are delivered within 15 seconds of establishment by the Tracking Function. The test case shall also generate tracks marked for output at various rates based on expected target loading on the system. In addition, rates of generation shall be used that stress the function beyond that specified to determine its ability to handle short term peak loading effects.

Software Top Level Design Allocation

The SRS requirements have been allocated to a computer software component named the Target Message Reporting Component (see Figure 5.2-3). For brevity, all aspects of the software top level design have not been included. The requirement for errored message retransmission has been used for the following example.

3.6.1 Target Message Reporting

3.6.1.1 Inputs

Errored Transmission Indicator from Communications Processing Function (1 word, maximum input rate of 1 per 100 transmitted messages)

3.6.1.2 Logical Data

3.6.1.3 Interrupts

SOFTWARE REQUIREMENTS	SOFTWARE TOP LEVEL DESIGN COMPONENTS AND ASSOCIATED TEST CASES
•	
•	
•	
3.2.8	
•	
•	
•	
3.2.8.3	3.6.1 / 4.4.218
•	
•	
•	

REQUIREMENT: PROCESS RETRANSMISSION REQUEST WITHIN ALLOCATED TIME

TEST CASE: VERIFY ABILITY TO PROCESS REQUEST WITHIN SPECIFIED TIME UNDER NOMINAL AND PEAK LOADING CONDITIONS

FIGURE 5.2-3: SOFTWARE REQUIREMENTS TO TOP LEVEL DESIGN COMPONENTS TRACEABILITY MATRIX

3.6.1.4 Timing and Sequencing

The Errored Transmisssion Indicator shall be processed within 300 milliseconds of receipt and shall output a request for retransmission to the Communications Processing Function within 350 milliseconds of completion of processing under worst case system loading conditions.

3.6.1.5 Processing

3.6.1.6 Output

Target Message Retransmission Request to Communications Processing Function (20 word, format as defined in Appendix A)

4.4.218 Test Case for Target Message Reporting Component

This test case shall verify that the Target Message Reporting Component is capable of processing the retransmission request within its allocated time. It will generate Errored Transmission Indicators at the specified rates and measure the time of generating the Target Message Retransmission Request. In addition, the test case shall generate Errored Transmission Indicators at rates exceeding those specified to determine its ability to process Errored Transmission Indicators under peak loading conditions exceeding those specified for the component. To measure the response time accurately, the measurements shall be conducted in an environment replicating or simulating actual operational conditions.

Software Detail Design Allocation

The software top level design has been allocated to a set of units that actually implement the process of responding to Errored Transmission Indicators (see Figure 5.2-4). One of the units accepts the Errored Transmission Indicator, decodes it, and initiates the processing by other units in the component. This control unit would be described in the following type format.

3.3.4.3.1 Errored Transmission Indicator Control Unit

3.3.4.3.1.1 Inputs

3.3.4.3.1.2 Local Data

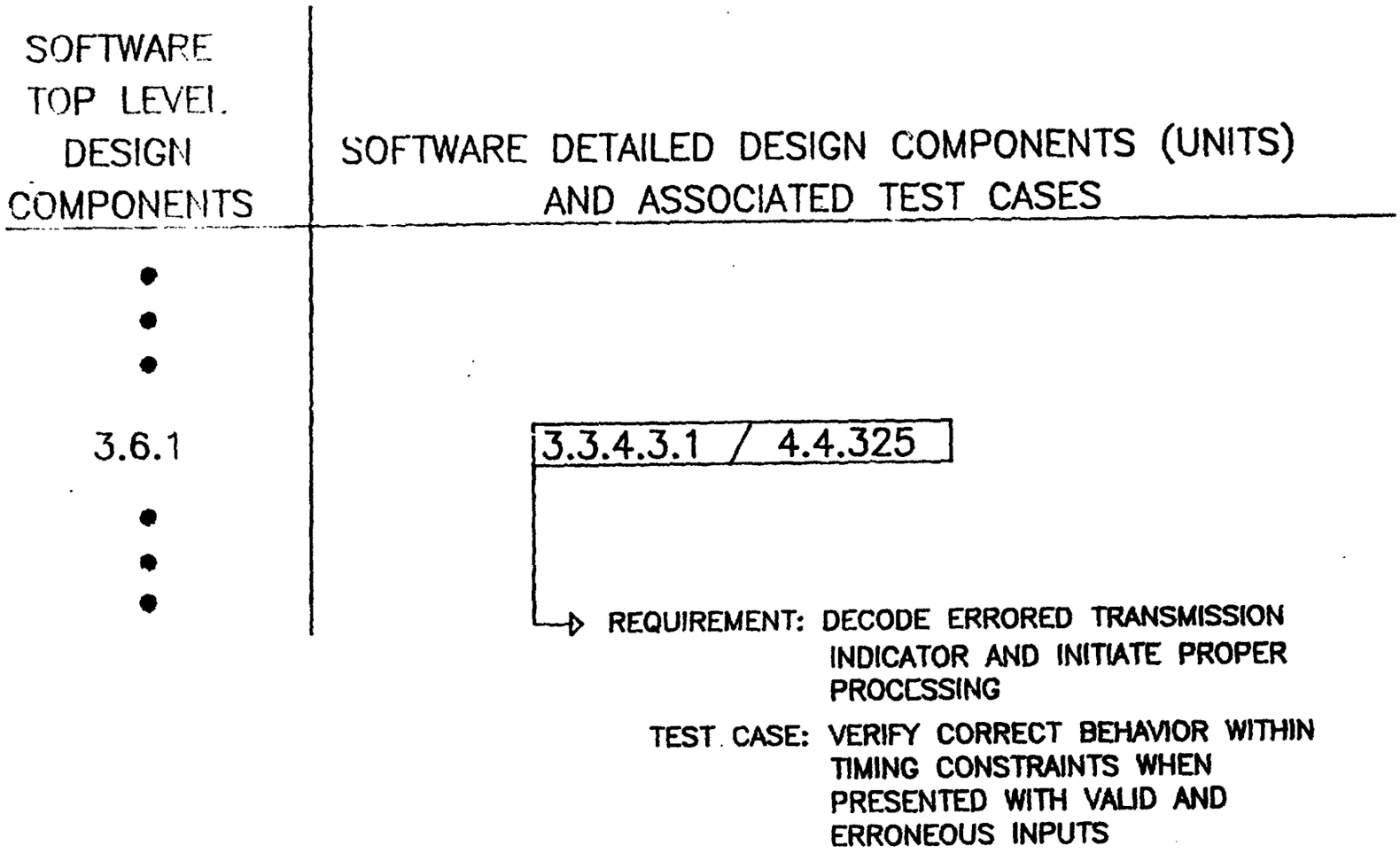

FIGURE 5.2–4: SOFTWARE TOP LEVEL TO DETAILED DESIGN COMPONENTS TRACEABILITY MATRIX

3.3.4.3.1.3 Process Control

3.3.4.3.1.4 Processing

3.3.4.3.1.5 Utilization of other Elements

3.3.4.3.1.6 Limitations

3.3.4.3.1.7 Outputs

4.4.325 Test Case for Errored Transmission Indicator Control Unit

This test case shall verify that the unit properly decodes the Errored Transmission Indicator and calls the units that process the message. It shall consist of proper and erroneous Errored Transmission Indicators, and shall determine whether the control unit properly queues requests, calls the correct units, and responds to errored requests. In addition, the test shall determine whether or not response to the requests occurs within the allocated time for the unit.

The traceability matrices illustrated above are essential tools for the management of the software development and test process. In addition to assuring completeness and revealing the introduction of non-required functionality, they can be used during the maintenance process to estimate the impact of changing or adding requirements. Finally, as shown above, they can be used to ensure adequate coverage of test objectives with respect to system and software requirements.

Selecting a Test Approach to Satisfy a Test Objective

In order to satisfy a test objective, a sequence of tests must be carried out. That is, a series of software executions must take place. These executions can be scenarios of the real, integrated system operating in its intended environment, or they can be single instances of simple program units (such as subroutines) operating in isolation from all other program units. The data that stimulates the software in any such test is referred to as "test data". During OT, the test data for the software is contained in the overall mission profile data used to define the test scenarios (e.g., the data defining the position of the airplane is determined by the actual position of the airplane during the test, the limits of resolution of any sensing devices, and the rate at which positional data is sampled). In laboratory settings, the test data may be supplied by simulators of the mission profiles. During unit and integration test phases, the data is supplied by programs or human testers that generate values for unit parameters, input variables, and other unresolved data references that the software may request. In each of these cases, the test data should be derived according to a specified technique or test methodology.

The principal requirement for test methodologies is that they support the corresponding test objectives. In many cases, the objective will specify the range of acceptable methodologies. In other cases, the methodology is selected as a result of an analysis of the objective and how to best support it. Two common errors in test planning are to either overspecify or underspecify the test methodology that supports the objective. Overspecification occurs when the test designer jumps directly into the detailed description of test procedures (e.g., "...move switch labeled 'BAY_2' to position 'ENABLED', observe blinking locator symbol at position...") without first defining the reason that this test supports the corresponding objective. Underspecification occurs when neither the test objective nor the test specified in the test plan constrain the tester to present test data that meets any special technical criteria. Such ad hoc testing is very difficult to use in a rigorous justification that the objective has been met. Ad hoc tests are also not sharable, and so should be avoided for that reason as well.

Matching methodologies to objectives usually involves cost-benefit and other tradeoffs. This is also a good opportunity to assess the realism of the test objectives. For example, an objective that can only be satisfied by an exorbitantly expensive test (the cost of which far overshadows its potential benefits) is probably not realistic and should be modified. The matching process is, to some extent, controlled by the degree of confirmation required by an objective. There are two approaches:

Necessary Conditions: A test methodology represents a necessary condition on the test if the only way in which the test objective can be satisfied is by the success of the test methodology. In other words, if the software fails to satisfy the conditions of the test, then the test objective has not been met.

Sufficient Conditions: A sufficient condition is one that implies the test objective. That is the test methodology presents a sufficient condition for the objective if the success of the methodology means that the objective has been satisfied.

For example, a certain computer program may have a test objective to demonstrate that 90% of all inputs are processed correctly. If there are only 10 possible inputs, then a sufficient test methodology is to exhaustively test all inputs. If all are processed properly, then the objective is satisfied. As an example of necessary condition, suppose that a certain system has a test objective to demonstrate that the system can perform in an operational environment for a period of ten hours with at most 25 failures to perform an essential function. A necessary condition, then, is that no sequence of test data to the software results in more than 25 mission-essential failures of a software function over the same period. A family of methodologies that frequently are used to provide necessary conditions are the so-called "coverage" tests. Generally speaking, test data satisfies a coverage condition if a proportion of a structural characteristic has been exercised by the test data. For instance, a specified target identification algorithm in a software component of an electronic warfare system may be implemented by a table in which the 10 possible target types are represented by rows of the table. The columns may represent the 225 distinct sensor configurations that are possible. An entry in the table represents the action to be performed when the indicated target is detected and the sensors are in the selected configuration. If a test objective is to demonstrate that all targets are correctly acted upon regardless of the sensor configuration, then it is necessary to structurally cover all 2,250 target-configuration table entries. Notice that this may not be sufficient to guarantee that the objective is met (there may be real-time constraints that cause correctly identified targets to be processed incorrectly, for example). However, if even one of the entries of the target-configuration table is not exercised by test data, then that specific action may lead to incorrect processing and so the objective of demonstrating that all target-configuration pairs lead to correct actions cannot be met.

One way of classifying the available systematic testing approaches is in terms of Test Type (static or dynamic), Visibility (black-box or white-box), and Life Cycle Phase (early or late). Static and dynamic test types differ in the extent to which software code is actually executed during the conduct of the test. When utilizing static analysis techniques, the software is usually not executed; rather, human code readers or automated analyzers process program text in order to resolve outstanding issues. Dynamic analysis techniques, on the other hand, depend on code execution for data concerning the software's behavior. Black-box and white-box tests are distinguished by whether or not internal program structure is used to define test cases or determine effectiveness of the tests. In black-box tests, the program is considered to be a closed system, about which only external input-output behavior is significant. White-box testing approaches are free to use all structural information to define the test. Finally, the details of applying a test approach are heavily influenced by the life cycle phase.

While it is, in principle, possible to consider all kinds of tests in this scheme, in practice, certain combinations of test types, visibility, and life cycle phase seem to work more effectively. For example, it is common to find static, white-box analyses used during very early design phases, but hardly at all during very late development phases or during operation and maintenance.

The following is a summary of systematic testing approaches that are commonly used to support test objectives [DeM 87].

Static Analysis

Static test methods are primarily used during early design stages to verify the consistency of intermediate engineering or incomplete software products with prior specifications or other documents. The nature of static methods makes them ill-suited for directly addressing operational issues. Nevertheless, static analysis is the principle tool for deriving early estimates of whether or not required technical characteristics are satisfied. Some of these methods simply measure the extent to which basic engineering standards have been satisfied. Still others carry out more sophisticated analyses (e.g., is there software which cannot be reached by any feasible control path?).

An important class of static methods involves the structured "reading" of program instructions. These readings or walkthroughs can be carried out in a group format (during which the group may play an adversarial role) or by a single programmer. Some major issues that can be resolved with this technique are the following: (1) Is the design complete? That is, does every specified requirement get successfully addressed in the design? (2) Is the design consistent with itself as well as with previous specifications and constraints? and, (3) Are all elements of the design traceable to a specific required function feature or capability? One difficulty in relying on static analysis, particularly those approaches utilizing subjective evaluations or human readers, is their relative non-reproducibility. Although such tests can be planned and carried out to support specific test objectives, test reporting and sharing of test results may not be possible.

Structural Coverage

A structural coverage test is satisfied if test data can be supplied that causes the execution of the specified percentage of the structural features of the software to be executed. A common structural coverage measure is statement coverage: a K% statement coverage test is satisfied by any test data that results in the execution of K% of the basic statements in the software program. Variations on statement coverage include such conditions as demonstrating that each executed statement is necessary. Another structural coverage measure is decision-to-decision branch coverage. This is a generalization of statement coverage. In statement coverage testing, a conditional or branching statement is considered the same as a non-branching statement: as long as it is executed by the test data, it contributes to the total coverage score of the test. For many programs that contain branches, however, the mere execution of a conditional statement is less significant than exercising all of the possible outcomes of the conditional. These decision-to-decision branches must be covered at the specified percentage in order to pass this kind of test. Coverage of higher level structural features (e.g., the entries of the target-configuration table discussed above) may also be specified in a test plan. While structural coverage tests are seldom used as sufficient conditions for a test, they are commonly included in test plans as necessary tests.

Domain and Path Coverage

It is common to use distinct processing paths in the software to partition the universe of possible inputs to the software into "domains". In the case that each processing path thus selected corresponds to a specified processing thread for one or more functions, then the distinct domains may also be interpreted as types of inputs as in the HUD processing example given above. A frequently useful methodology can be based on simply specifying the proportion of the total number of paths that must be covered by the test data. This is not, however, a simple structural coverage test since these paths can involve the complex intertwining of structural features (e.g., the multiple and interleaved repetition of loops). Like the structurally-based tests, simple path coverage methodologies tend to be useful in generating necessary but not always sufficient tests. In some programs, the number of paths is so large that the proportion of the paths is not meaningful. In these cases, the "highly likely" paths may be selected for execution. If these paths correspond to distinct types of inputs about which statistical frequencies are known, then path coverage tests may be useful for developing sufficient conditions on the test objectives.

Frequently, the domains have a mathematical or geometric structure that can be exploited. The "domain strategies" use previously gathered information about the likelihood of certain kinds of errors involving the definition of domains to generate tests. When depicted graphically, these tests might specify that a certain subset of the test points should be selected on one side of a domain boundary, a second subset on the other, and at least one test point should fall exactly on the domain boundary. If such rules guarantee that certain kinds of domain errors will be revealed (if present) then the domain strategy may be an effective necessary test.

Another type of path coverage test that is often used to infer that objectives are satisfied is the "symbolic test". In symbolic testing, a mathematical formula is derived by executing a certain path or set of paths symbolically. This formula is then compared to the specified or required properties of the path. If these are consistent, then the path represents a correctly implemented series of processing steps.

Tests for Computational Errors

Even when statements have been executed and domains have been selected properly, there are many errors that arise due to mistakes in designing computational steps. A useful test methodology for detecting computational errors is "data flow analysis". This is actually a family of analysis methods that detect anomalies in the handling of data that are indicative of errors. Data flow analysis will detect, for example, situations in which a data value is accessed by the software before a definite value has been assigned to it. Data flow analysis measures are easily automated and may be included in the compiler for a modern high order programming language such as the Ada* language.

A particularly intractable kind of computational error is one that defeats a test because the test result is coincidentally the same as a point at which the computation happens to give correct results (e.g., if a test results in calculating 2x at the point x=1, and the program should have calculated 2 raised to the power x, the test will not detect the error). No single computational methodology is sufficient to detect such errors. As a result, the tester may have to apply a variety of test methodologies.

Error Coverage

Tests that specify error coverage criteria are passed when test data are supplied that demonstrate that the given errors do not occur in the software. If the errors have a relationship to the test objectives, then error coverage tests are useful sufficient tests of some kinds of objectives. As a methodology for ruling out various kinds of errors, at least minimal error coverage tests are usually necessary.

A common category of error coverage tests are mutation tests. These are the software version of the single fault coverage tests frequently used for digital hardware. In mutation tests, the errors are modeled on a statistical sampling of the errors that the programmers are likely to make. Because of its statistical basis, mutation testing can be tailored to a given set of test objectives.

* Ada is a registered trademark of the U. S. Government (Ada Joint Program Office).

Other error coverage tests (related to mutation) include variations of path and domain testing in which the selection of test data for each path and domain is guided by a set of rules that will reveal certain types of errors if they are present and the "error-sensitive test case analysis" and "weak mutation" methods which provide for fixed test data construction rules that will uncover the occurrence of specified errors.

Incremental and Non-Incremental Integration Tests

A common objective in integration tests is to demonstrate the accumulated increase in functionality gained as software units are bound together in subsystems and systems. The performance of the integrated components is specified by external and functional specifications. Sometimes these specifications can be hierarchically decomposed and identified with individual units as shown in Figure 5.2-5. In this case, incremental integration tests may provide both necessary and sufficient test criteria for the stated objectives. These tests demonstrate the correct implementation of each of the decomposed subfunctions and then verify that the integrating components combine the subfunctions properly.

Frequently, however, the functions specified at the subsystem level do not have an obvious incremental relationship to the software units. In this case, a non-incremental test such as a "thread test" is needed. In these tests, processing paths or threads are identified and tests are designed to support the objectives associated with the threads. The distinction between incremental and non-incremental tests is that during incremental tests, higher level tests combine the tests of their components. During non-incremental tests, on the other hand, external process descriptions (such as might be contained in a system verification diagram or software requirements definition) are used to generate the tests which might involve the intertwining of several units.

Functional Tests

Functional tests demonstrate the correct implementation of functional specifications and requirements. These may be either operational or technical specifications. While many functional testing methodologies are specific to the application at hand, several general concepts recur.

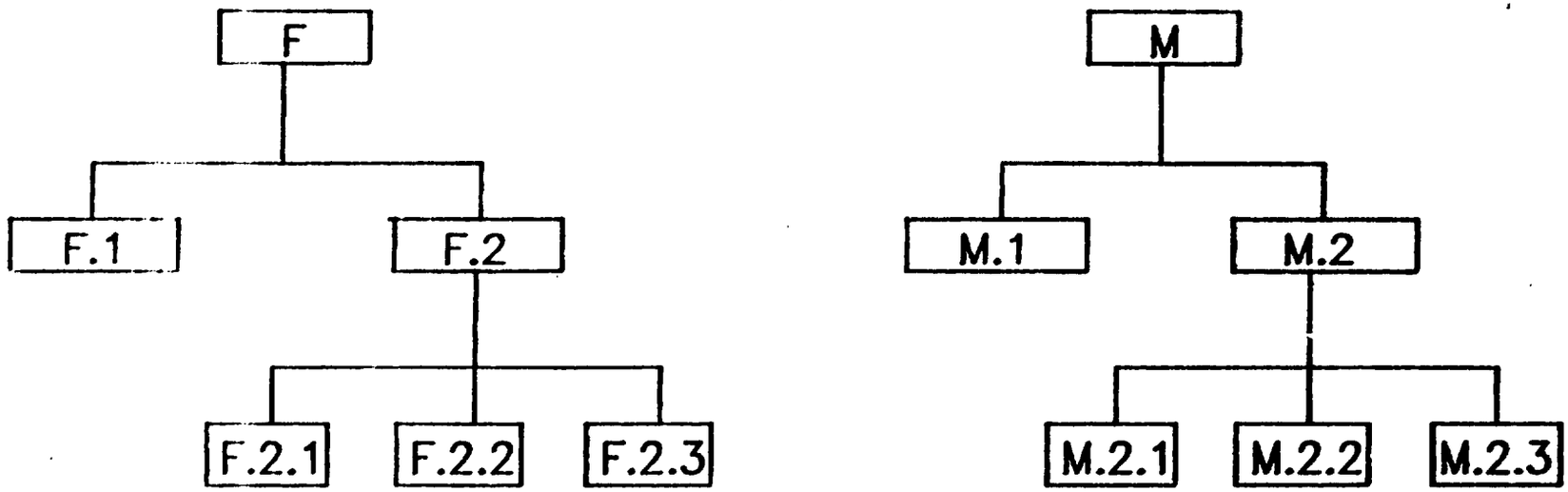

FIGURE 5.2–5: SUBSYSTEM M IMPLEMENTING FUNCTION F

A common goal of functional testing is to stress the functions. That is, to demonstrate the behavior of the software when limits and capacities are approached and exceeded. At the unit level, this may involve selecting tests that correspond to extreme or "out-of-spec" values for unit parameters or input variables. At the system or subsystem level, the corresponding stress test may involve test data that saturates a given system capability. During OT, functional stressing can be demonstrated by exceeding software system capacities by specified amounts. While stress testing is seldom sufficient to determine whether or not objectives have been met, these functional tests are frequently necessary to exhibit the performance parameter limits of the system and to exhibit failure modes and effects.

Another useful functional test methodology is "random" testing. Random tests involve the selection, generation, or extraction of test data from a statistical or stochastic source. During unit testing, the statistical source may be a generator that samples from a source according to a certain probability distribution. During integration testing, a simulator may provide the statistically meaningful frequencies of occurrence of data values, while during OT, the statistical variations in the actual mission profiles are the source of randomness. If the test designer has confidence in his knowledge of the underlying probability distributions, then random testing can be used to effectively estimate operational reliabilities and other statistical parameters. In these instances, random tests are used as sufficient conditions on the test objectives.

Late Life Cycle Testing

The applicability of systematic test approaches during late life cycle phases such as operation and maintenance is determined more by external factors than individual characteristics of the test approaches. For example, using an error-based test for in-line testing of software in an integrated, operational system is usually possible provided that the necessary instrumentation is available and there is enough free processor capacity to support the overhead associated with the test.

An important class of late life cycle tests is the regression testing that must be carried out to assess the impact of software changes. Key issues of regression tests revolve around the expense of re-running large numbers of individual tests. While these concerns are influenced by project management strategies, some technical aids are useful. Being able to assess the impact of program changes requires two-way traceability between program text and the historical record of critical T&E issues. Furthermore, errors, failures, and corrective actions should be reported and tracked to enhance traceability.

Testing Real-Time or Embedded Software

Embedded, real-time and parallel applications often involve software test issues that require extra attention. Many of the methodologies that apply readily to sequential applications become infeasible in these environments. Often the only available methodologies require that test objectives be scaled down. A major source of difficulty in testing software for these applications is the complexity of interactions between components. Whereas sequentially organized programs can be tested and test results described by deterministic methods, parallelism includes stochastic behavior and multi-threaded control. The sheer number of possible system states often frustrates thorough tests of integrated systems. In some cases, system design can influence the ease of testing (e.g., loosely coupled parallel systems are easier to test than tightly coupled ones). The early involvement of testers in the design process is valuable in steering designers away from inherently untestable system architectures. In other cases, early testing of operational aspects of system components can be used in place of OT objectives for the system as a whole.

In many cases, instrumentation also presents special problems when conducting these and related tests. During early test phases many properties of the running programs may be invisible to the tester without instrumenting devices such as counters that indicate which branches and paths have been executed. During tests of integrated software systems, it becomes even more difficult to peer into dynamic aspects of the software. For example, in the HUD processing application discussed earlier, the success or failure of a software function may depend on the rate at which a certain software buffer fills and empties. During unit testing this buffer may be completely visible as an array or some other data structure in a given unit. In an integrated system running on an operational hardware set, it is usually very hard to directly observe such dynamic aspects of software performance without instruments that are tailored to the task. The alternatives are generally labor-intensive and are frequently not cost-effective to implement. For example, a common way of gaining visibility into such detailed features of the software is to interrupt an OT so that the entire contents of the computer's memory can be "dumped" to magnetic tape or disk. This dump is then analyzed to extract information about the state of the software. Clearly this procedure cannot be used very often during the course of a test. On the other hand, it is frequently possible to place "software hooks" into delivered software so that hardware and software instruments can be conveniently attached during testing. If, for example, the message format of a communications program is likely to become a critical T&E issue, then it may be desirable to require the software developer to provide a software switch that, in test mode, causes a given message to be decomposed into a standard representation and output to a designated I/O port.

It has been found that, when instrumentation requirements are recognized early in a program, included in test plans, and included in the software requirements specifications, the quality of testing of major weapon systems is improved [STE 86].

Determining the Appropriate Level of Test

Most systematic test methodologies involve the selection of a level to which the test can be carried out. For example, in structural coverage methods, the level of the test is an explicit parameter: a 100% coverage of statements is a more thorough test than a 50% coverage. In specifying the level to which a test is carried out, the test plan should strive to resolve the relevant T&E issue with an appropriate margin of safety. In general, the higher the level of the test, the higher its cost. The lower levels of the same test have generally lower fidelity. These tradeoffs should appear as part of the explicit analysis during test planning.

It is frequently the case that some of the costs of selecting very high levels for a test can be offset by using a higher degree of capitalization for the test; that is, by using more and more powerful computers, reusable hardware and software instrumentation, or other capital expenses, the direct cost of which can be amortized across several tests.

Using DT&E to Demonstrate Operational Characteristics

There is not always a single, synchronized schedule for demonstrating test objectives. In particular, many T&E issues that involve operational characteristics can be demonstrated relatively early in the software development process. Since early demonstrations are almost always desirable, these opportunities should be actively sought out.

In some programs, this has resulted in the construction of nearly-operational integration test facilities for use during DT&E. Simulators of operational scenarios and stimulators which provide software inputs in place of external sensors and other devices can, of course, be used in these environments to reduce the costs or risks to life that would be incurred in an OT. Such facilities are particularly useful in demonstrating properties of the man-machine interface and measuring other human factors of the design. Software interfaces are usually very well-suited to these early tests since the appearance of functions at the interface can be completely designed and the corresponding functions can be simulated by hardware and software drivers, scripts, and simulators that are indistinguishable by a human operator from the real, operational functions. The use of string test facilities and environment simulators has been found to improve the overall success of software test programs [STE 86].

These facilities may have secondary benefits to the program. For instance, integration test facilities that demonstrate human factors of the man-machine interface, also tend to be good vehicles for training and may thus continue to be useful long after integration tests have been concluded.

5.3 Reporting Test Results

It might be supposed that in a properly planned test, the reporting of test results is a simple matter. In fact, test reporting is often complicated by the limitations of the test, ambiguous events that are noted during the test, and failures of instrumentation. This subsection presents some guidelines that are useful in reporting test results.

Identifying Discrepancies between Test Articles and Planned Operational Software

Test evaluations, particularly those aimed at operational issues, usually include assessments of how closely the test environment matches the planned operational environment. In the case of software, the report of test results should specify any discrepancies between the software being tested and the operational software. The impact of these discrepancies on the test objective being demonstrated should also be included, if it is known. The following are common sources of discrepancies.

Hardware Differences

Differences between test hardware and operational hardware can have a profound impact on the performance of the software. Differences in memory size and organization, instruction execution speed, processor and instruction set architecture can all affect the software. Except in rare circumstances, OT objectives should be satisfied on operational hardware.

Operating System Differences

Run-time operating systems found in typical operational environments are different than those used by programmers and designers. Run-time systems are usually optimized for performance and intolerant of significant deviations from specified usage. Operating systems used during development are usually optimized to allow great latitude on the part of the programmer and to facilitate sharing of resources such as memory and I/O devices. This means that capabilities demonstrated in the development environment may not be present in the operational environment. Any T&E issues that involve operating system primitives, resources, or calls to system functions should be redemonstrated in the operational environment.

Standard Library Calls

Many programming languages and environments implement standard operations such as mathematical functions and input-output in library packages that are supposed to be "standardized". However, due to differences in computer word sizes, underlying processor architectures, or any of a dozen other factors, the performance of these libraries can vary widely from one system to another. Therefore, if the operational software is required to compute the natural logarithm of a real number and does so through a library function, the accuracy of the returned value may have to be revalidated in the operational environment, even if the accuracy was determined perfectly in the development environment. Standard library calls should not be incorporated into operational software systems without addressing the possible differences. Standard sets of benchmark data may be available from third parties for comparing libraries.

Relating the Test Environment to the Operational Environment

Regardless of the care taken in structuring the circumstances of a software test, the limits imposed by the test itself mean that the test environment and the operational environment have differences. In laboratory testing of air defense systems, for example, killed targets must be removed from the screen by software. In the operational version of the same system that software is not necessary. The following are some common test limitations that give rise to discrepancies.

Simulated Inputs

The software may be stimulated in a much different manner in its operational environment than in its test environment. A simulator for real-time and embedded applications may not faithfully model timing conditions at the software level. Furthermore, actual software behavior may depend on converting analog events to digitized form in statistically unpredictable ways. Simulators, on the other hand almost always choose a fixed set of statistical models from which to generate digital stimuli to the software.

Friendly Users

During unit and integration tests and to some extent during system level development T&E (DT&E), the users of the software may be particularly friendly. They may know, for example that typing too fast on the keyboard causes the software to "get lost"; therefore, they will not do it. Operational users cannot be expected to be friendly. Operational T&E (OT&E) issues should be resolved with typical operational users.

The Maintenance Environment

Determining whether or not the software meets its maintenance requirements is best accomplished by using typical maintenance personnel in the environment where the actual maintenance will be carried out. Developers may have access to a host of special tools and capabilities that enable them to perform maintenance actions. Some of these may be proprietary or not otherwise deliverable with the operational software. Thus, using development personnel in the development environment is usually not a good way to resolve maintenance issues.

Interpreting Test Results in System Terms

In reporting test results it is important to address both the test objectives and critical T&E issues. If the resulting assessment deals with operational issues, the test results should be stated in system oriented terms.

Error Classification

It is especially important that software's contribution to operational suitability be properly reported. During laboratory tests and OT, the classification of test events such as those that require maintenance actions is an important aspect of test reporting [STE 86]. These classifications are analyzed to assign probable causes of failure and pinpoint system elements that contribute to high failure rates, system unavailability, or other attributes that negatively affect system suitability.

Figure 5.3-1 shows the usual method of classifying anomalous conditions (e.g., failures that arise during tests). Any incident during the test that requires unscheduled action to bring the system back to an operational status is an unscheduled maintenance action. These include failures of both hardware and software components. Not all of these incidents are classified as mission failures. Test reports should specify those failure conditions that impair the system's ability to perform an intended critical function.

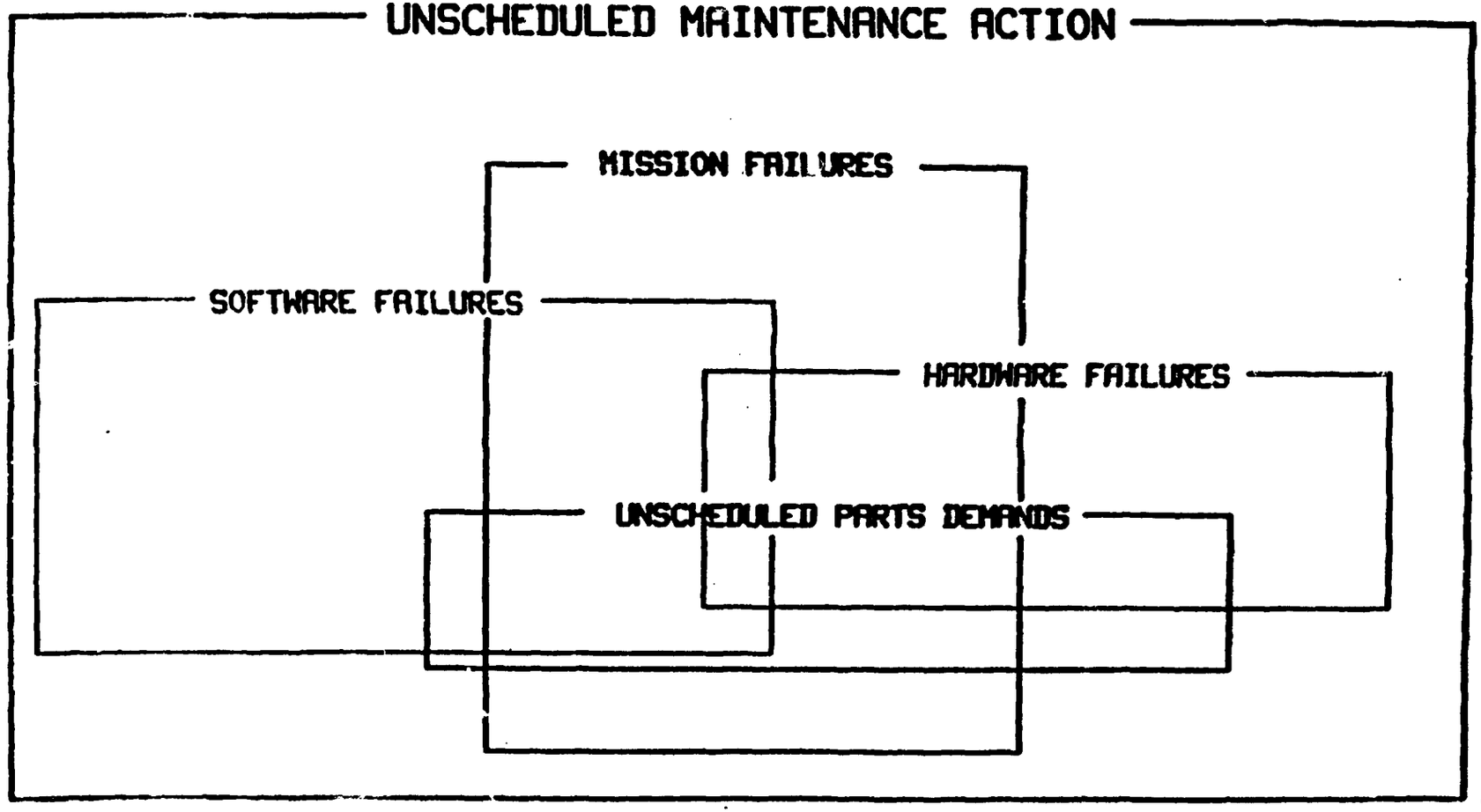

FIGURE 5.3–1: RELIABILITY INCIDENT CLASSIFICATION

Each software incident that involves a system function F should be classified by specifying the failure effect, cause, and required maintenance action. Early agreement in test plans on a common set of criteria for classifying reliability incidents is desirable. Common specifications include the following.

Effect.

1. F does not respond to stimulus or operator action.

2. F responds to stimulus but does not respond as intended.

3. F responds as intended to stimulus but is misinterpreted by the operator.

4. F responds to stimulus but with results degraded below stated threshold.

Cause.

1. Stimulus is outside F's stated operating range or requirements.

2. Failure is caused by mechanism external to components implementing F.

3. Failure is caused by mechanism internal to components implementing F.

Required Maintenance Action.

1. The effect of the failure is reversed without maintenance action.

2. The effect of the failure is reversed by action of an operator to restore the system to a non-failed state.

3. The effect of the failure is reversed by the action of a maintenance operation that changes the software.

4. The effect of the failure cannot be reversed by any maintenance action.

Downtime Classification

Figure 5.3-2 shows the preferred method of classifying system downtime during tests. This scheme should also be applied to the software incidents as specified above. These downtime assignments are particularly useful in testing for availability. The following examples illustrate some possible event classifications based on observed downtime classifications of software systems (see Figure 5.3-3).

Fault Tolerant Recovery. In this case, suppose the software detects the failed state and recovers, thereby placing the system in an operational state without operator intervention. The downtime classification for this event is the following: (1) operating -- the time up to the point at which the fault occurs, (2) standby -- the time during which recovery takes place and the failed function is no longer available, (3) operating -- the time interval commencing after full recovery of the function takes place.

Responding to Changing Requirements. This represents a maintenance action in which a change in software requirements necessitates a re-engineering of a portion of the system. The corresponding downtime classification is the following: (1) operating -- this is the period of time until the system is removed from operation because the changed requirements render the software ineffective in meeting system objectives (this interval may overlap substantially with the maintenance action of redesigning the indicated software component), (2) logistic delay -- this is the period during which re-installation of the redesigned system takes place, (3) operating -- the time interval commencing after the re-engineered software becomes fully operational.

Repair of a Fault. This action may occur at a remote maintenance facility and have the same downtime profile as the one given above if it is scheduled. If there is no logistic downtime (i.e., if the event is unscheduled) then the downtime classification is as follows: (1) operating -- the interval ending at the point that the failure renders the system inoperative, (2) active downtime -- the interval during which the failure is corrected by maintenance action (the system is expected to be functioning during this interval, but it is not), (3) operating -- the time interval commencing when the failure has been repaired and the system is restored to operational status.

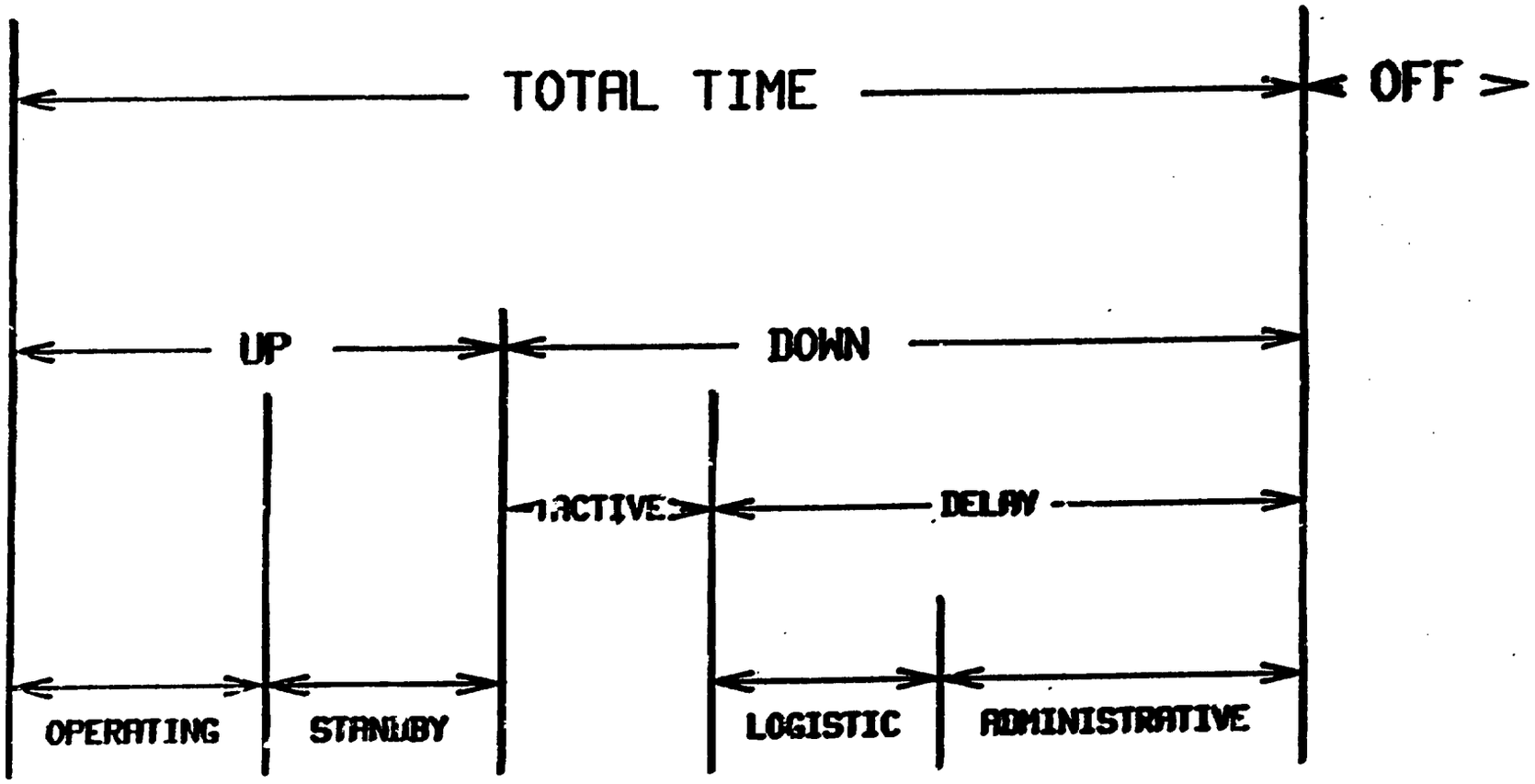

FIGURE 5.3–2: DOWNTIME CLASSIFICATION

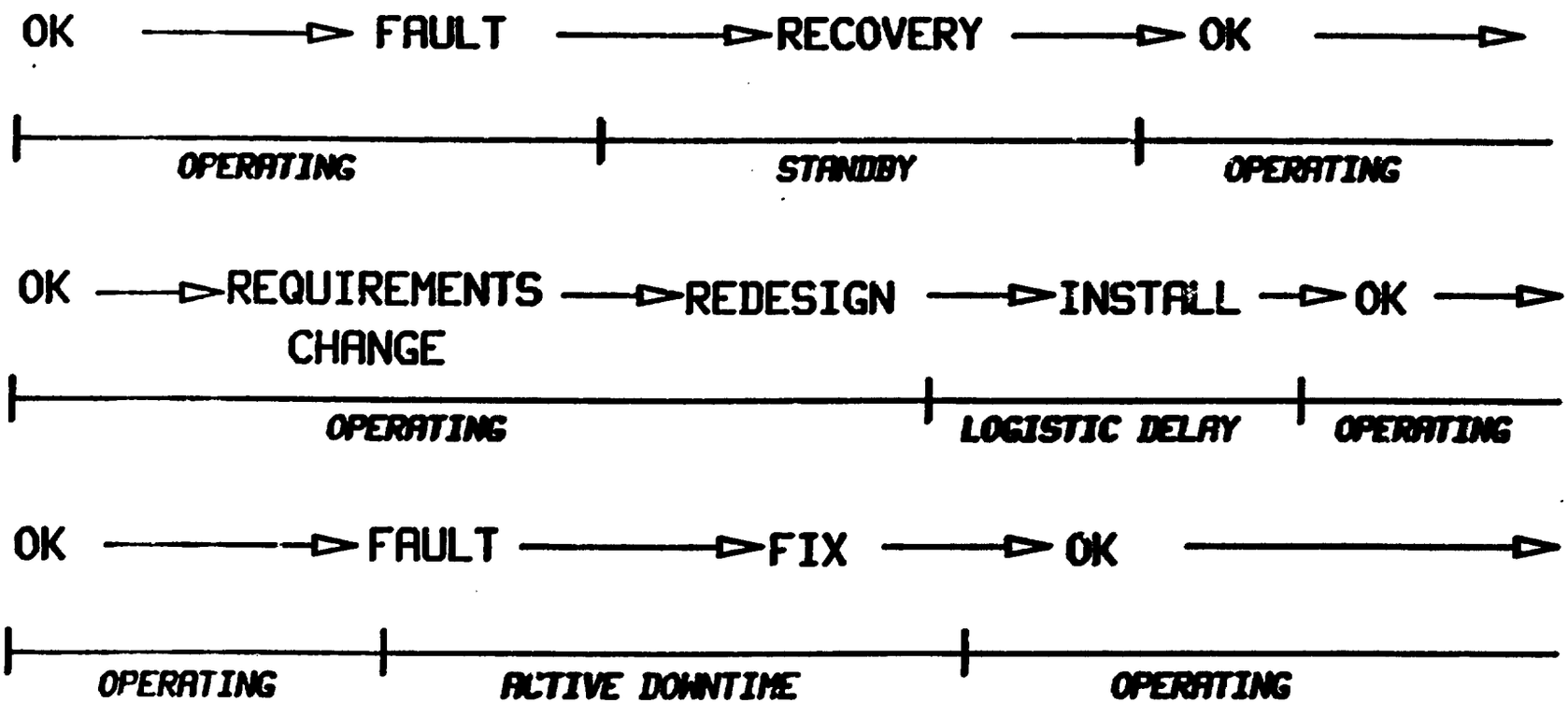

FIGURE 5.3–3: SOFTWARE EXAMPLES

Assessing Software Contributions to System Capabilities

The ultimate goal of reporting test results is to support the assessments of system level evaluators. Since these assessments are supposed to determine the contribution of software to risk, visibility into critical aspects of software design and implementation is an essential aspect of test reporting.

Test reports are not evaluations. Whether or not a test objective has been satisfied is determined by a separate analysis of the test results. The role of the tester is to independently measure and report the behavior of the system with respect to stated criteria. The evaluation process consists of a progressive assessment of risk based on the test results contained in test reports, supplemented with information concerning the extent of testing. The evaluator is ultimately responsible for assessing whether or not critical T&E issues have been resolved and to what extent unresolved issues contribute to overall engineering risk.

If the software development process has been structured and managed so that there is a progressive flow of intermediate engineering products and so that continual T&E of these products has been used to assess suitability and effectiveness, then evaluations can proceed in an orderly fashion. If these products are not available, an evaluation is extraordinarily difficult to carry out. The key issue here is visibility. In an integrated system consisting of large quantities of hardware and software, critical aspects of the software design and operation are effectively hidden under layers of software-software and hardware-software interfaces. During unit testing, for example, it is generally possible to use structural coverage and functional tests to thoroughly exercise a unit's input variables. In the integrated system, these variables may be receptacles for data filtered through several hardware and software subsystems. Thus, stressing an individual software component, as may be required to test a performance or effectiveness characteristic, may be very hard simply because the "real" independent variables of the test objective (the input variables for the unit) are now being controlled by other subsystems. Achieving a given status or condition of the unit may be dependent upon achieving values of the subsystems' inputs that are not possible within test limitations.

The evaluation criteria for each of the intermediate engineering products should reflect the sequential decomposition of test objectives. At the highest level of the evaluation hierarchy are those indicators and criteria that are used to judge the overall status of the test program. Volume I of this series of manuals elaborates these criteria. Below this level, the evaluation criteria should be structured so that they support one or more criteria from the level directly above. The most effective method of developing such criteria is to follow the approach outlined in Section 3.3 for top-down decomposition of test issues. As each test issue is resolved on the basis of meeting or failing to meet a specified test objective, the evaluation criteria for that test should attempt to aggregate the test results into an overall picture of software system status until the topmost level of the hierarchy (i.e., the system level criteria) has been addressed.

CHAPTER 6

SOFTWARE TEST TOOLS AND RESOURCES

Many of the testing methodologies described in Section 5.2 can only be applied with the support of an automated tool. Without automation, the possibility of human error during the application of a testing methodology may negate any "guarantees" associated with that methodology (e.g., error coverage methodologies can only guarantee the absence of specific errors when they are supported by correctly implemented tools). In addition, the computational demands imposed by many of the methodologies make them impossible to implement manually. This chapter will outline areas of importance to the specification of automation requirements, provide pointers to sources of information about existing automated tools, and discuss risks to be considered during the selection of software testing tools.

6.1 Specifying Requirements for Automated Support

DoD 5000.3-M-1 defines test resources as follows:

> Test Resources. A collective term that encompasses all elements necessary to plan, conduct, and collect/analyze data from a test event or program. Test funding is the most important test resource since all other resource elements are derived from it. Other elements include test conduct and support manpower (including TDY costs), test assets (or, units under test), test asset support equipment, technical data, simulation models, testbeds, threat simulators, surrogates and replicas, special instrumentation peculiar to a given test asset or test event, targets, tracking and data acquisition instrumentation, and equipment for data reduction, communications, meteorology, utilities, photography, calibration, security, recovery, maintenance and repair, frequency management and control, and base/facility support services.

The most important software test resources are automated software testing tools. The primary requirement that must be satisfied by any automated testing tool applied on a program is that it support the defined test objectives. For example, if a low level test objective is to exercise 85% of all decision-to-decision branches in specified units of software, then an appropriate software testing tool would be one that instruments the software and reports execution statistics. If another test objective is to demonstrate that a critical algorithm can complete processing within 2 milliseconds when the system is operating under heavy load conditions, the capability to measure timing statistics must be available.

It is unlikely that a single testing tool will be able to support all test objectives for a program. In most cases, a variety of testing capabilities will be needed. One approach to satisfying this requirement is the acquisition or composition of a toolset to support testing activities. Although this may currently be the most feasible alternative, there are major pitfalls associated with this approach. The most critical barrier to the effective use of a set of testing tools is the fact that unless the tools were specifically designed to cooperate, it is very unlikely that they will do so without significant effort on the part of the tool user or tester. Even in cases where inter-tool information exchange is not a concern, it is desirable that the tools work on a common representation of the software. Again, this cannot be assumed. An alternate approach is the acquisition or development of a testing environment designed and built to maximize cooperation between elements, minimize proliferation of software representations, and ease the addition of capabilities. Although implementations may not be readily available, recent efforts have defined the functional capabilities necessary to comprise a testing environment to support the needs of large programs [DeM 86]. Regardless of the packaging, however, the guiding principle when selecting automated software testing capabilities is that they should not be employed unless they specifically support a carefully designed test. Otherwise, their use will not contribute to the overall goals of the test program.

Individual automated software testing tools can usually be classified as either static analyzers, dynamic analyzers, or test support tools. The remainder of this section will discuss the range of functionality that can be found in these classes of tools.

Static Analysis Tools

Static analysis tools, or static analyzers, examine information that can be obtained from a piece of software without actually executing it. This includes information pertaining to the component's structure, and design or coding conventions. Existing static analyzers vary greatly in both scope and functionality, ranging from systems which enforce standards to those which perform sophisticated analyses. Functions implemented in static analysis tools are usually selected from the following set [DAC 85, DeM 87].

Code Auditing: The determination, or enforcement, of adherence to established procedures or standards.

Completeness Checking: The determination of whether or not all subcomponents necessary to form a component are present and fully developed.

Consistency Checking: The determination of whether or not each component is internally consistent in the sense that it contains uniform notation and terminology, and externally consistent with respect to its specification.

Cross Reference: The referencing of entities to other entities by logical means.

Data Flow Analysis: The graphical analysis of sequential patterns of data definitions and references to determine constraints that can be placed on data values at various points of execution.

Error Checking: The determination of discrepancies, their importance, and/or their cause.

Interface Analysis: The inspection of the interfaces between software components for consistency and adherence to predefined rules and/or axioms.

I/O Specification Analysis: The examination of a software component's input and output specifications, usually for the purpose of generating input data.

Type Analysis: The determination of correct use of named data items and operations, and whether or not the domain of values attributed to an entity are properly and consistently defined.

Units Analysis: The determination of whether or not the units or physical dimensions attributed to an entity are properly defined and consistently used.

Dynamic Analysis Tools

Dynamic analysis tools, or dynamic analyzers, are tools that support testing by collecting and examining information resulting from the direct execution of the software being tested. Dynamic analyzers can be further classified as symbolic evaluators, test data generators, program instrumenters, or program mutation analyzers. Each of these classes of tools and the functions they perform will be discussed below.

Symbolic Evaluators

Symbolic execution is a verification technique that simulates software execution using symbols rather than actual values as input data [DAC 85]. The execution output consists of logical or mathematical expressions with these symbols representing the component's variables. Tools that implement symbolic execution are usually called symbolic evaluators. One significant function performed by these tools is constraint evaluation which consists of simplifying path input or output constraints and determining their consistency with pre-existing constraints. This function is also implemented in some test data generators.

Test Data Generators

A test data generator is a tool that aids in the generation of test data for software components [DeM 87]. This relieves the tester of much of the tedium involved in the generation of large amounts of test data and avoids the introduction of bias into the test set when the tester is, in fact, the software developer. It should be noted that the test data generator can only partially automate the construction of test cases, since a test case consists of both test data and expected output. The expected output is usually determined by hand calculation or simulation.

The three primary types of test data generators are pathwise, data specification, and random test data generators. Pathwise test data generators create input data that is a comprehensive representation of the input space by selecting input data from the input domains associated with the software's paths. Data specification systems provide a data specification language that the tester uses to describe the input data. The system then uses the description to generate the desired input data. Random test generators select random points from the domain of each input variable of a component. For the randomness to be meaningful, it must be applied to both the selection of data within a path domain and the selection of the path domains.

Program Instrumenters

Program instrumenters insert monitoring statements, or probes, into the source code of the software component under test to gather execution data that reveals detailed information concerning the software's internal behavior and performance. Functions performed or supported by existing program instrumenters are usually selected from the following set [DAC 85, DeM 87].

Assertion Checking: The evaluation of user-embedded statements in a component that assert relationships between its elements. An assertion is a logical expression specifying a program state that must exist or a set of conditions that the variables must satisfy at a particular point during software execution.

Resource Utilization Analysis: The gathering and evaluation of system hardware or software resource utilization statistics.

Structural Coverage Analysis: The determination of measures associated with the invocation of the component's structural elements to determine the adequacy of a test run. Coverage analysis is useful when the user is attempting to execute each statement, branch, path, or iterative structure in a software component.

Timing Analysis: The estimation or measurement of execution time of a software component either by summing the execution times of the instructions in each path, or by inserting probes at specific points in the software and measuring the execution time between probes.

Tracing: The construction of a record of all or certain classes of instructions or events occurring during execution of a software component.

Tuning: The optimization of system/software performance.

Mutation Testing Tools

Mutation testing tools support test data entry, execution, and error coverage analysis for the purpose of determining the adequacy of the test data based on the results of program mutation [DeM 87]. Existing mutation testing systems provide an interactive test environment, and reporting and debugging operations that are useful for locating and removing errors.

Test Support Tools

Test support tools perform a variety of functions such as test execution coordination, simulation of unavailable inputs or components, or regression testing [DeM 87]. Automatic test drivers, also known as test harnesses or testbeds, are software systems that provide an environment for running software component tests and simulating missing components or subsystems. They provide a standard notation for specifying test cases, automate the verification of test results, and eliminate the need for writing separate drivers and stubs for unit and subsystems testing. Simulators support testing by representing the circumstances of nominal and stressful operational scenarios, or other computer systems or software not available for testing purposes. Regression testing systems support the selective retesting of modified software to detect faults introduced during the modification process, or to verify that changes have not caused unintended effects and that the component still meets its specified requirements. Other tools of use during the testing process are data collection and reduction tools, report generators, and error tracking tools.

Once the desired automated testing capabilities have been defined, decisions must be made with respect to the system hardware and software configurations that can be utilized for testing. For each desired capability, the following requirements must be defined. First, the target language for the software testing tool must be specified (i.e., the programming language used to implement the portion of software to which the relevant test objective applies). Second, the hardware and operating systems available for the execution of the tests must be defined. Finally, the resources available (e.g., memory, storage, throughput) for the purposes of testing must be estimated. These requirements will be used as selection criteria and constraints during the evaluation of available software testing tools.

6.2 Determining Tool Availability

The four primary sources of software testing tools are government organizations, government contractors, commercial organizations, and academia. The approach used to locate tools within each community differs and is described below (see Appendix D for points of contact).

Communication within the Military Services and other government organizations normally follows well-defined chains of command. Thus, when trying to identify existing software testing tools, the first point of contact would be the individual or group within the Headquarters organization that is responsible for MCCR or embedded computer resources. The next point of contact would be within the Development Commands. From there, references to individual Program Offices, Logistics Groups, or Laboratories would be expected.

An exception to this (soon to be the rule) occurs if the target language of interest is the Ada language. In that case, the Ada Joint Program Office (DUSDRE(R&AT)) is the logical starting point in a search for testing tools. In addition, other government agencies that employ the Ada language (e.g., Defense Intelligence Agency (DIA), Defense Nuclear Agency (DNA), National Security Agency (NSA), Defense Advanced Research Projects Agency (DARPA), Defense Communications Agency (DCA)) may also provide opportunities for automated support of testing.

Pointers to government contractors can be gained by soliciting references from the government contacts. Inquiries directed at industrial organizations (e.g., National Security Industrial Association (NSIA), Electronics Industries Association (EIA)) or examination of their conference proceedings can also prove fruitful.

Commercial vendors can be expected to advertise in tools catalogs such as Data Sources. Surveys reported in industry periodicals such as DATAMATION and DATAPRO are also sources of information about existing commercial products.

Academic enterprises are usually the subject of publications in journals and presentations at conferences sponsored by groups such as the Institute of Electrical and Electronic Engineers (IEEE) or the Association for Computing Machinery (ACM). In addition, since university researchers in the areas of computer science and software engineering communicate extensively over national networks, examination of network bulletin boards and users' groups' newsletters can provide information not available elsewhere, including critical reviews of capabilities.

Finally, the Data and Analysis Center for Software maintains a Software Life Cycle Tools Directory [DAC 85] that includes entries describing testing tools available from all sources listed above.

It should be noted that not all tools discussed in public forums are necessarily available, at any price, for public use! In situations where a tool provides a competitive edge, its developer may label it as proprietary. Even though it is used during the development testing of a software system, it will not be delivered for use during the maintenance of that system and other arrangements will have to be made for regression testing.

In the event that a tool cannot be acquired or developed to support the objectives of the overall test program, the software test planning process must be revisited and the adequacy of the modified test plans evaluated. In some cases, it may be sufficient to determine and document the reduced level of confidence in the test due to the required changes. In other cases, however, a significant redesign of the test may be unavoidable.

6.3 Assessing the Risk of Using Selected Test Tools

Although the process is not as straightforward as that of purchasing a spreadsheet package for a personal computer, it is possible to locate and acquire software testing tools. The available tools differ greatly with respect to maturity, size of intended user group, level of support, extent of documentation, restrictions on usage, and price. Knowing whether a tool was developed as a commercial venture, custom built, developed in-house, or the product of a laboratory prototyping activity can temper expectations. Typical attributes of tools resulting from each of these development paradigms will be described below.

Commercially built tools are usually the most mature of the four development types listed above. This is due to the intent to market the tools to a wide audience and provide support for the tools while they are in use. In this case, a company's profits depend on the quality and usability of the tools. The need for customer satisfaction with respect to usability implies a greater likelihood of the existence of training materials and documentation for the tools. In this case, it is not unreasonable to request references (i.e., names of individuals who have experience applying the tools on a real project) so that relatively unbiased assessments of a tool's effectiveness and suitability can be considered during the decision-making process. Since these tools are commercial products, restrictions on distribution and modification usually exist.

Custom built tools will not usually have been subjected to the same degree of examination prior to delivery as commercially available products. In addition, since all costs associated with development must be borne by a single customer, their prices are usually significant. For the investment, the customer can expect a tool that satisfies a specific customer need as opposed to a widespread generic need. The degree of documentation, training aids, and support provided with custom built tools depends very much on the customer's willingness to support these items. Although it seems apparent that the customer would own unlimited rights with respect to tools usage, this cannot be assumed; care must be exercised during negotiation of the tool development contract.

Software testing tools built "in-house" greatly resemble custom built tools. However, one potential hazard associated with their development must be stressed. With the custom built tools described above, it was assumed that a legal contractual agreement would specify the requirements for the tool's functionality, documentation, training aids, support, and acceptance. When the tool development is conducted within the customer organization, the danger always exists that resources will be shifted with the changing demands and atmosphere of the company business. Rather than the tool benefiting from a concentrated effort, it may be treated as a toy to be worked on during the development personnel's spare time. The development of a software testing tool is a software development effort and must be subjected to the same discipline as any other development if similar quality achievements are to be expected.

The final type of tool development to be addressed here is that of tools resulting from efforts conducted in a laboratory. These tools are usually prototypes built for the purpose of feasibility assessments or proof of principle investigations. Very seldom are they considered to be "industrial-strength". They may contain numerous errors, suffer from inefficient implementation, and have little or no supporting documentation or training information. The benefits of laboratory tools are that they usually implement state-of-the-art testing methodologies and, in some cases, can be obtained for a token fee. Although it is not wise to assume that a laboratory tool can be applied to a project "as-is", if resources are available to productize and support the tool, it can be a good investment.

In cases where the software testing tools are to be supported by the using organizations, the determination of tool suitability must be concerned with its maintainability. Issues of importance include the implementation language of the tool, the availability of development documentation as opposed to usage documentation, and the employment of modern software engineering practices during the tool's development. These issues are even more important if there are plans to tailor or add new capabilities to the tool after acquisition. If future plans include porting the tool to a new hardware/software configuration, examination of the tool prior to acquisition should also be concerned with issues such as the isolation of machine dependencies in the design and the implementation.

Chapter 6: Software Test Tools and Resources

A primary requirement for the implementation of a sound software test program is the acquisition of software testing tools to support defined test objectives. However, the use of these tools may introduce risks to the development project. First, there are technical risks introduced by unproven tools that must be eliminated by testing to validate their correctness. Second, schedule risks are concerned with the tool being available for the planned software testing; the allotment of sufficient time for the training of personnel using the testing tool; the proper amount of time being allowed for the actual testing to occur; and the consideration of testing alternatives in the event that the testing tool is not available. Third, budget risks address the hidden costs that are introduced due to the resources needed for the execution of the testing tools; the training of personnel for using the software testing tools; and the maintenance of the tools. The awareness that these risks may accompany the use of software testing tools should not discourage their use but ensure that proper management and planning for the use of automated technology are incorporated in the software test program.

APPENDIX A

LIST OF ACRONYMS

COTS	Commercial Off-the-Shelf
DCP	Decision Coordinating Paper
DID	Data Item Descriptions
DoD	Department of Defense
DoDD	Department of Defense Directive
DT	Development Test
DT&E	Development Test and Evaluation
HUD	Heads Up Display
IOC	Initial Operational Capability
IPS	Integrated Program Summary
ISO/OSI	International Organization for Standardization/Open Systems Interconnection
IV&V	Independent Verification and Validation
JRMB	Joint Resources Management Board
MCCR	Mission Critical Computer Resources
OSD	Office of the Secretary of Defense
OT	Operational Test
OT&E	Operational Test and Evaluation
OTS	Off-the-Shelf
QA	Quality Assurance
SCP	System Concept Paper
SLOCs	Source Lines of Code
SRS	Software Requirements Specification

SSA	Software Support Agency
SSS	System/Segment Specification
STD	Software Test Description
STP	Software Test Plan
STPR	Software Test Procedure
STR	Software Test Report
T&E	Test and Evaluation
TEMP	Test and Evaluation Master Plan

APPENDIX B

REFERENCES

[Alf 77] Alford, M. W., "A Requirements Engineering Methodology for Real Time Processing Requirements," IEEE Transactions on Computers, January 1977, pp. 60-69.

[Boe 76] Barry W. Boehm, "Software Engineering," IEEE Transactions on Computers, December 1976, pp. 1226-1241.

[Boe 81] Barry W. Boehm, Software Engineering Economics, Prentice-Hall, Inc., 1981.

[DAC 85] Data & Analysis Center for Software, "Software Life Cycle Tools Directory," IIT Research Institute, March 1985.

[DeM 86] Richard A. DeMillo, "Functional Capabilities of a Test and Evaluation Subenvironment in an Advanced Software Engineering Environment," Software Engineering Research Center Report No. GIT-SERC-86/07, Georgia Institute of Technology, October 1986.

[DeM 87] Richard A. DeMillo, W. Michael McCracken, R. J. Martin, and John F. Passafiume, Software Testing and Evaluation, Benjamin/Cummings Publishing Company, 1987.

[E&V 84] "Evaluation and Validation Plan," Version 2.0, Prepared for the Ada Joint Program Office, Wright-Patterson Air Force Base, Ohio, December 1984.

[EIA 84] Electronics Industries Association, Requirements Committee, Government Division, "Analyses and Forecasts of Specific Markets: DoD Computing Activities & Programs," December 1985.

[Orl 84] Orlando I Software Workshop, Panel B, "Independent Verification and Validation," Final Report of the Joint Logistics Commanders' Workshop on Post Deployment Software Support (PDSS) for Mission-Critical Computer Software, Volume II - Workshop Proceedings, June 1984.

[Red 84] Samual T. Redwine, Jr., Louise Giovane Becker, Ann B. Marmor-Squires, R. J. Martin, Sarah H. Nash, and William E. Riddle, "DoD Related Software Technology Requirements, Practices, and Prospects for the Future," Institute for Defense Analyses Paper P-1788, June 1984.

[Ros 77] D. Ross, and K. Schoman, "Structured Analysis for Requirements Definition," IEEE Transactions on Software Engineering, January 1977, pp. 6-15.

[STE 86] Software Test and Evaluation Project, "Software Test and Evaluation Manual, Volume III, Good Examples of Software Testing in the Department of Defense," Software Engineering Research Center Report No. GIT-SERC-86/06, Georgia Institute of Technology, October 1986.

[Tei 77] D. Teichroew, and E. Hershey, "PSL/PSA: A Computer Aided Technique for Structured Documentation and Analysis of Information Processing Systems," IEEE Transactions on Software Engineering, January 1977, pp. 41-48.

APPENDIX C

DEPARTMENT OF DEFENSE DIRECTIVES AND STANDARDS

DoD Directive 5000.1, "Major System Acquisitions," March 12, 1986.

DoD Directive 5000.3, "Test and Evaluation," March 12, 1986.

DoD 5000.3-M-1, "Test and Evaluation Master Plan Guidelines," October 1986.

DoD-STD-2167, "Defense System Software Development," June 4, 1985.

APPENDIX D

POINTS OF CONTACT

Software Life Cycle Tools Directory

Data & Analysis Center for Software (DACS)
RADC/COED
Griffiss AFB, New York 13441
(315) 336-0937
Autovon 587-3395

Catalogs

Data Sources
P.O. Box 5854
Cherry Hill, New Jersey 08034

DATAPRO
1805 Underwood Boulevard
Delran, New Jersey 08075
(609) 764-0100

Ada Tools

Ada Information Clearinghouse
Rm. 3D139 (Fern St./C-107)
The Pentagon
Washington, DC 20301-3081
(703) 685-1477

Industry Periodicals

DATAMATION
875 Third Avenue
New York, New York 10022

Industrial Organizations

Electronics Industries Association
2001 Eye Street, N.W.
Washington, D.C. 20006
(202) 457-4900

National Security Industrial Association
Suite 901, 1015 15th Street, N.W.
Washington, D.C. 20005
(202) 393-3620

Professional Societies

Association for Computing Machinery
1133 Avenue of the Americas
New York, New York 10036

IEEE Computer Society
Post Office Box 80452
Worldway Postal Center
Los Angeles, CA 90080